

The book cover features a dark background with a vibrant rainbow on the left side. A bright, glowing white oval is positioned at the top center. Below it, a view of the Earth from space is visible. A series of blue lines radiate from the bottom center towards the horizon, creating a perspective effect. The title 'Software Theory' is written in a blue, italicized serif font within the white oval. The subtitle 'for Parallel Processors' is written in a white, italicized serif font below the main title. The authors' names are listed in a white sans-serif font at the bottom.

Software Theory

for Parallel Processors

W.C. Cave, R.E. Wassmer, H.F. Ledgard, K.T. Irvine, & E.S. Slatt

Software Theory[®]

for

Parallel Processors

by

W.C. Cave, R.E. Wassmer, H.F. Ledgard, K.T. Irvine, & E.S. Slatt

March, 2016

© Copyright 2013, 2014, 2015, 2016 by

VISUAL SOFTWARE INTERNATIONAL

309 Morris Avenue

Spring Lake, New Jersey 07762



(732) 449-6800



VSI@VisiSoft.com



(732) 449-0897



www.VisiSoft.com

TABLE OF CONTENTS

SECTION	PAGE
ACKNOWLEDGEMENTS	v
PREFACE	vii
Chapter 1 - INTRODUCTION	1-1
Chapter 2 - UNDERSTANDING THE PROBLEM	2-1
Chapter 3 - THE EVOLUTION OF COMPUTERS	3-1
Chapter 4 - THE EVOLUTION OF SOFTWARE	4-1
Chapter 5 - BASIC PRINCIPLES APPLIED TO SOFTWARE	5-1
Chapter 6 - UNDERSTANDING PARALLEL PROCESSING	6-1
Chapter 7 - MODELS AND SPACES OF SYSTEMS AND SOFTWARE	7-1
Chapter 8 - AN INTEGRATED SOLUTION APPROACH	8-1
Chapter 9 - SOFTWARE ARCHITECTURES FOR PARALLEL PROCESSING	9-1
Chapter 10 - SOFTWARE LANGUAGES FOR PARALLEL PROCESSING	10-1
Chapter 11 - THE LANGUAGE OF MEMORY RESOURCES	11-1
Chapter 12 - THE LANGUAGE OF PROCESS INSTRUCTIONS	12-1
Chapter 13 - THE TOP LEVEL CONTROL LANGUAGE	13-1
Chapter 14 - GENERATING A TAILORED RUN-TIME SYSTEM	14-1
Chapter 15 - VPOS - A PARALLEL PROCESSING OPERATING SYSTEM	15-1
Chapter 16 - IMPROVING PARALLEL PROCESSOR HARDWARE DESIGN	16-1
Chapter 17 - SINGLE PROCESSOR TESTS & RESULTS	17-1
Chapter 18 - PARALLEL PROCESSOR TESTS & RESULTS	18-1
Chapter 19 - OPTIMIZING DESIGNS FOR PARALLEL PROCESSING	19-1
Chapter 20 - NONSTATIONARY APPLICATION ARCHITECTURES	20-1
REFERENCES	R-1
APPENDICES	A-1

ACKNOWLEDGEMENTS

The authors would like to acknowledge the many contributors who provided ideas, assistance, and material that led to this book. This includes the development of key concepts, historical back up material, and the technology described here. The total list of people who have contributed is much too large to provide here and likely would not be complete.

The essence of this technology dates back to the 1960s when a small group of electrical engineers led by Bill Cave created a Computer-Aided Design (CAD) system for electronic circuit design. Bob Wassmer joined this group in 1968. Prediction Systems, Inc. (PSI) was formed by Cave in 1974, and soon joined by Wassmer, to create and market the General Stochastic Modeling (GSM) system. This simulation facility automatically generated sets of nonlinear differential equations and fast solutions for modeling complex decision systems. This was followed in 1982 by the creation of the General Simulation System (GSS) for modeling communication and control systems using discrete event simulation. The GSS system, with its underlying requirement for parallel processing, is the genesis of the technology described here.

Many of the direct contributors are former as well as current employees of PSI. Since 1982, argumentation, experimentation, trial and change, using first hand experience on what works best - and what doesn't, were an overwhelming part of the culture at PSI. Without that special culture to get it 'right' or 'best' from a productivity standpoint, including the willingness to try new approaches developed by others, this technology would not exist.

We acknowledge those who made significant contributions, many of whom worked directly on the software, some as users shaping the approach. Much of the early software was developed by Dave Furtzaig, Pat Miller, Marty Bopp, and Kim Martis. Early in-house users who provided design guidance were Richard Guilfoyle, Hwa-Feng Chiou, Zong-An Chen, Don Lin, Dana Alexe, Rachel Marrotta, Luke Nitti, Jose Ucles, Ken Saari, Sanjiv Sukumaran, and Bill Wollman. Naeem Malik was instrumental in helping to install systems and train users in Europe as well as the U.S. John Fikus provided inputs as a user and developer of complex modules, 3D graphics, and material for describing what GSS does.

Famous programming language designer and critic Henry Ledgard worked on consulting contracts with Bill Cave in the mid-1970s when software productivity was considered important. This led to the design of much of the language environment of GSS. After moving to the University of Toledo (UT), Henry introduced PSI to Shunqian Wang from UT, who eventually managed the development team. Shunqian brought students from Tongji University in Shanghai, e.g., Lin Yang, Yanna Jaing, Mingzhou Yang and Lei Shi. Shunqian's leadership was especially critical to the development of the graphical software, both in the CAD interface facilities, and with the run-time graphics.

In the early 1990s, it became clear to the development team that the technology used to build models and simulations in GSS was far superior to that being used to build the software underlying GSS. It was also clear that GSS contained everything needed to build software as well as special simulation facilities. This led to the development of the Visual Software Environment (VSE) - the software environment equivalent of GSS.

Development of many of the underlying concepts of VSE would not have been possible without the first hand knowledge and analytical judgments from Bob Wassmer. He has had overall responsibility for both the GSS and VSE development team since the earliest beginnings, contributing as the key manager over all the years of this effort, as well as major designer and user.

Ken Irvine has been involved in the concepts and basic design of GSS and VSE from a user standpoint as well as in developing the software itself. He has always questioned anything that appeared to be off-track, and has provided great help with our written material in general, as well as this book. Ed Slatt, being a significant user building large scale planning tools, has provided contributions in the more esoteric area of parallel processing, as well as general requirements.

Dave Hendrickx and Manu Thawani have played key roles since the very early years, helping with analysis of design approaches as well as development of extremely complex modules and software. As an outside consultant and user, Dave was the GSS User's Group Chairman for over a decade.

Finally, contributions came from large numbers of clients without whom none of this would exist.

PREFACE

While computer clock speeds were doubling every 18 months (from about 1982 to 2006), software speeds should have doubled accordingly. Yet application improvements did not realize the speed benefits due to the poor approach to software development. As clock speeds leveled off, computer manufacturers have put multiple processors (cores) on a chip and multiple chips on a board. This might lead one to believe that using 8 processors could increase the speed by a factor close to 8. This assumes that all processors are being used efficiently to support an application - something that eludes current software approaches.

Sufficient inherent parallelism must exist in the application system, else processor utilization efficiency will be low causing low speed multipliers. More importantly, inherent parallelism must be properly mapped onto parallel processors. This requires subject area experts who understand the application. Expecting compilers to do this, e.g., tiling of loops, wastes time unless loops are sufficiently large and independent, a rare case. Expecting the operating system to map the inherent parallelism is folly, especially if the application system is nonstationary.

Solving the concurrency problem in parallel processing implies translating the inherent parallelism in a system into a software architecture that runs correctly on a parallel processor. This implies that parallel processor solutions must be complete and consistent with those produced on a single processor.

Effective mapping of inherent parallelism requires designing large hierarchical data spaces. This requires a language that supports their use as well as design. With the proper language facilities, large data spaces can be copied in a single instruction fetch, with all data elements directly available. This implies copying blocks of memory - a critical speed factor. Memory is cheap and abundant; saving it creates bottlenecks and wastes time.

Misunderstandings of the problem are historic. Software developers have been unable to cope with parallel processors for decades, putting most hardware manufacturers out of business. More recently they have moved the problem back on the hardware designers, causing precious chip space to be wasted on irrelevant solutions. Recent statements by the top technical officers at Intel, Microsoft, and AMD say that current computer languages cannot handle multi-core / parallel processor applications. They have made an urgent call for changing the way people build software, saying it is time for a new approach, see [47], [84], [99], [152]. But resistance to change is immense and well documented in other technology fields, see [46], and [85]. It is due to: (1) fear for job security; (2) huge investments in existing approaches, and (3) the Not Invented Here (NIH) syndrome.

For decades, experts have dropped from the field, complaining that software is far from a scientific technology, and not based upon experiment. Journal articles rarely compare data representing measures of improvement, such as time to run an application, or time to build and test it. Many articles fall into the category described by Bailey, [8], "Twelve Ways To Fool The Masses When Giving Performance Results On Parallel Computers." Without overcoming the huge resistance inhibiting fair comparisons, a good solution to the parallel processing problem will never be tried, let alone accepted. The main purpose of this book is to provide a scientific foundation for the technology of software.

Software theory follows from Shannon's *Mathematical Theory Of Communications*, also known as Information Theory. The basis for this theory is that the binary number system, the foundation for modern computers, forms a mathematical space wherein the general set of characters used to write software and control devices is represented by strings of bits or binary numbers.

Another concept underlying this theory is the State Space framework, formulated by engineers to extend Control Theory in order to solve complex problems associated with the design of Control Systems. A prime example is controlling the orbits of objects in space. This theory extends the mathematics of vectors and matrices into more complex transformations that deal with sets of large vector spaces.

The extension of this theory is described in Chapter 7, where the concept of Generalized State Space is introduced. This requires two additional concepts. The first is derived from the requirement for completeness and consistency of results when using parallel processors, i.e., that two processes must be *independent* in order to run concurrently on two different processors (as used here, a process is a sequence of machine instructions). To be independent, these processes must not share data.

This leads to the next concept that, to simplify the determination of the independence of two processes, one must separate data from instructions at the software language level. Known as the Separation Principle, this provides the ability to represent data structures and rule structures (processes) using icons on engineering drawings of software. We note that engineering drawings represent the *connectivity* of elements; they are not flow charts. Engineering drawings of software provide an iconic visualization of what processes share what data (memory resources), and therefore their independence. By grouping these icons into hierarchies of modules, the drawings represent transformations on the state vectors.

Since the state vector elements are represented by binary numbers and may contain general character data, they are termed *Generalized State Vectors*. The transformations may contain IF ... THEN ... ELSE statements and are termed *Generalized Transformations*. This framework for designing software is called *Generalized State Space*.

We note that this framework was developed in 1982 to support the design of the General Simulation System (GSS). GSS was developed as a Computer-Aided Design (CAD) tool to simplify the design of discrete event simulations of communication and control systems on parallel processors. Since then it has evolved into VisiSoft, a CAD system for building complex software as well as simulations for parallel processors.

Having refined the approach described here over many years, and having compared it to current approaches to building software on many different projects it is apparent that, without a scientific basis, one will never take real advantage of parallel processor technology. More importantly, parallel processing hardware technology is now headed in questionable directions - to make up for the lack of science in the current approaches to software.

CHAPTER 1. INTRODUCTION

As one gets older, one becomes more appreciative of time. This is particularly true if one is trying to accomplish specific objectives, advance the state-of-the-art, or demonstrate new technologies. Nowhere is the importance of time more prevalent than in the computer field.

The first *all-electronic* computer was developed to reduce the time needed to solve the equations required to design the Atomic Bomb. It replaced hundreds of people with hand calculators. This was followed by the Hydrogen bomb, a much more difficult problem. The need for greater speed drove development of the *stored-program* computer. The major hurdle was providing sufficient memory to handle larger sets of numbers as well as stored programs.

In the 1960s, the rule was that the cost of internal memory would never fall below 10 cents a bit. Today, one can buy a 32 Gigabyte memory for \$20 (10 cents buys 1.28×10^9 bits). Since then, engineers have tackled the problem of building faster computers, breaking down the computational speed barriers with dramatic increases in both clock rates and memory.

Shortly after the new Millennium, the great driving force behind increasing computer speeds was predicted to come to a halt. This was the ending of the part of Moore's rule that computer clock speeds would double every 18 months. This has been borne out by the leveling of clock rates at about 4 Gigahertz. As this was becoming a reality, computer manufacturers started to put multiple processors on a chip, saying that speeds should continue to improve as the number of processors in a single computer climbed to unforeseen heights. However, software providers have tried to make good use of parallel processors since the early 1960s. Except for special "embarrassingly parallel" problems, this has been difficult to achieve.

In addition, many software applications have become more complex. Increasing complexity is generally the barrier to growth when advancing the state of technology in any field. Without the ability to linearize growing complexity, it grows exponentially - limiting the ability to move ahead. Conquering complexity requires breakthroughs in approach. This is clearly true in the case of software. Fortunately, the other half of the speed driver, memory, is becoming more abundant and cheap. Using more memory has again become the key to speed.

Technological breakthroughs are almost always the result of science. In the end, improvements and breakthroughs are measured against prior results. The thesis of this book is that software breakthroughs and improvements are measured in terms of time. As indicated above, this implies the time to develop as well as time to run a piece of software.

The purpose of this book is to describe a theory of software that can be proven by test data taken from repeatable experiments designed to make fair time and speed comparisons. This testing is best initiated using stop watches to measure run-times. Our experience is that, following run-time experiments where the outcomes are obvious, one is prone to make fair comparisons of the development times (productivity) as well. The rest is left to science.

The authors of this book have been developing Computer-Aided Design (CAD) tools for engineering since the 1960s. To maximize ease-of-use of the CAD tools, one must deal with complexity - behind the scenes. The CAD tool described here makes parallel processor software easier to develop than current approaches for a single processor. To make this CAD system so easy to use, the underlying design is complex. Without a CAD tool for building this extremely complex software, the final CAD tool itself would have never been completed.

The Importance Of Science in Advancing Computers And Software

Computers have played a major role in improving the lifestyle and chances of survival of the human race. This may be born out simply by looking at the dramatic increase in availability of information to anyone with a computer. Without computers and the sophisticated software behind them, the design of most everything we depend upon today would be difficult, and in many cases nonexistent. These are the underlying forces that drive the desire for (1) faster computers; and (2) improved productivity in creating and running applications software. These improvements are measured by comparing the time it takes to build and run important applications.

With the advent of the integrated circuit chip, the problem moved from computer hardware to software. During the 1960s and up until the 1970s, great strides were made in computer languages, driven by measurable improvements in speed and productivity.

This all changed during the 1970s when knowledgeable software language designers were replaced by corporate power battles and PR. The major difference in the popular software world of today is the lack of a scientific approach used to evaluate run-time speed and productivity. This mind-set has now moved into the realm of hardware design. Today, the lack of science in software is leading hardware designers astray.

Software is rapidly becoming the world's largest and most important industry. It now provides the backbone facilities underlying many new technologies.

The Hydrogen Bomb was not created based upon the popular tastes of the time or the styles that appealed to its designers. It was derived to produce a very clear result based on scientific experiments and measurements.

Applying A Scientific Approach To Parallel Processing

To apply a scientific approach to building software for parallel processors requires the following steps:

- Gain an in-depth understanding of the application.
- Create a decomposition of the application system based upon its inherent parallelism.
- Map the inherent parallelism of the application into a software architecture of independent modules that can be visibly inspected using engineering drawings.
- Generate the code for complex algorithms that all members of the development and support team can easily understand.
- Optimize the allocation of parallel processor resources to the independent modules.

Performance of the above steps requires a fully integrated approach to the use of parallel processors. It requires a development environment where designers can easily create optimized software architectures. The development environment must automatically generate a run-time system with dynamic allocation facilities that are tailored to the hardware environment as well as the software architecture. The run-time system must support dynamic reallocation of parallel processor resources when run-time connectivity of the architecture is stochastic.

The Basis For Real Science

Many scientists contributed to the development of Nuclear power. They include people like Bohr, Einstein, Fermi, Heisenberg, ..., etc. Each contributed theories backed by experimental evidence that could be used to advance the required breakthroughs to get to the final tests. These efforts took many years, as the underlying theory evolved. When the theory was mostly in place, it took a large team of many of the best scientists another few years to complete the development and testing of the first Atomic bomb. It took another decade after that to complete tests of the Hydrogen bomb. It was this last decade, following World War II, that led to the development of the *first stored program computer*, the Maniac, used to support the Los Alamos Laboratories in New Mexico in development of the Hydrogen bomb.

Real science is based upon repeatable experiments and the measurements derived from them. The data taken from these experiments is used to uncover the truth about competing theories. It also leads to new unanticipated theories. It is then shared among scientists so that further advances can be made. Clearly, it has been this truth-seeking approach to science, and the corresponding advancements in technology, that have brought the human race to where it is today.

The scientific method is governed by the ability of different people to repeat the experiments of the originator. Only unbiased people questioning the theory can take the measurements - independently - and reveal the data that supports their findings.

As described by Lord Kelvin,

“When you can measure what you are speaking about, ... you know something about it; but when you cannot measure it, ... your knowledge is of a meager and unsatisfactory kind ...”

From W. Edwards Deming, Father of Quality control, [53].

“If you can’t measure it, you can’t improve it.”

From Anselmo & Ledgard, Communications of the ACM, [2].

“We take for granted our ability to compare hardware productivity using benchmarks and purchase hardware based upon them. There are no acceptable productivity benchmarks for a software environment. Comparisons are generally based upon literature advocating a given method. Invariably they lack scientific data to support the claims.”

From Henry Ledgard, Communications of the ACM, [89].

“The field has yet to measure the productivity of competing software development environments. The software industry deserves some objective investigations in this area. In today’s situation, one can say that, without measures from repeatable experiments, software is not a science.”

The approach to software described here is based upon many years of hard science inquiry, competing theories, scrutinized logical deductions, experiments, measurements, and resulting data. Both the development time and run time of different applications can be measured against the clock.

Comments From Engineers - on current approaches to building software:

The comments immediately below by Rattner (Intel CTO), Moore (AMD Senior Fellow, Technology Group CTO, and Chairman of the Technology Advisory Board), and Mundie (Microsoft Chief Research & Strategy Officer) indicate the level of urgency toward solving the software problem while talking about the number of years it will likely take based upon history. Of this group, Chuck Moore (recently deceased) had a clear understanding of the problem and undoubtedly would have fully appreciated the VisiSoft solution presented here. It's apparent from the articles that barriers to a totally new approach to building software must be overcome.

[EE Times](#): ESC Fall 2007 Preview: Multi-cores, software's Gordian Knot, see [47].

To fully utilize the hardware parallelism inherent in embedded multi-core designs, they say, will require a shift to a more implicitly parallel programming language and methodology. However, many, including researchers at Microsoft, believe that it will take at least ten years for the industry to shift to a new parallel programming framework.

[EE Times](#): Intel CTO presses software developers to keep pace, see [84].

Software development and delivery have failed to keep pace with advances in computer hardware, according to Intel Corp.'s CTO, (Justin) Rattner: -- "As hardware technology approaches the terascale level on the desktop, software has fallen further behind." -- "One result has been a lack of parallel programming applications to leverage dual-and multi-core processing technology. Intel is looking for 'new languages for programming in parallel,' Rattner told the India Semiconductor Association.

[EE Times](#): Industry seeks a model for next-gen multicore CPUs, see [98].

"The industry is in a little bit of a panic about how to program multi-core processors, especially heterogeneous ones," said Chuck Moore, a senior fellow at Advanced Micro Devices trying to rally support for work in the area. "To make effective use of multi-core hardware today you need a PhD in computer science. That can't continue if we want to enable heterogeneous CPUs," he said. --- The challenge in the parallel world is finding a dynamic and flexible approach to schedule parallel tasks from these modules across available hardware in complex heterogeneous multi-core CPUs.

[EE Times](#): Multicore puts screws to parallel-programming models, see [99].

Leaders in mainstream computing are intensifying efforts to find a parallel-programming model to feed the multicore processors already on chip makers' drawing boards. --- Developers need to expand the current software stack in fundamental ways to handle a coming crop of processors that use a variety of cores, accelerators and memory types, according to the company. --- Both AMD and Intel have said they will ship processors using a mix of X86 and graphics cores as early as next year, with core counts quickly rising to eight or more per chip. But software developers are still stuck with a mainly serial programming model that cannot easily take advantage of the new hardware. --- Thus, there's little doubt the computer industry needs a new parallel-programming model to support these multicore processors. But just what that model will be, and when and how it will arrive, are still up in the air.

[Reuters](#) - Craig Mundie, Microsoft Corp's chief research and strategy officer, is sure he has a good handle on where technology is going. When is another story, see [152].

The computer industry has taken its first steps toward parallel computing in recent years by using "multi-core" chips, but Mundie said this is the "tip of the iceberg." --- To maximize computing horsepower, software makers will need to change how software programmers work. Only a handful of programmers in the world know how to write software code to divide computing tasks into chunks that can be processed at the same time instead of a traditional, linear, one-job-at-a-time approach. --- A new programming language would be required, and could affect how almost every piece of software is written. --- "This problem will be hard," admitted Mundie, who worked on parallel computing as the head of supercomputer company Alliant Computer Systems before joining Microsoft. "This challenge looms large over the next 5 to 10 years."

Software Productivity

Since the late 1980s, productivity in software has declined - *faster than any other industry in the U.S.* (see charts below). Whereas computer chips had the highest productivity growth of all industries in the 5 years prior to 1995 (chart 1), computer software had the lowest. Software actually had negative productivity growth (red bar). The high cost of building and maintaining software, and large number of project failures has put many projects on hold. Large companies are now outsourcing software projects overseas to India, China, and similar countries.

CHART 1. Data From Business Week - January 1995 [26]

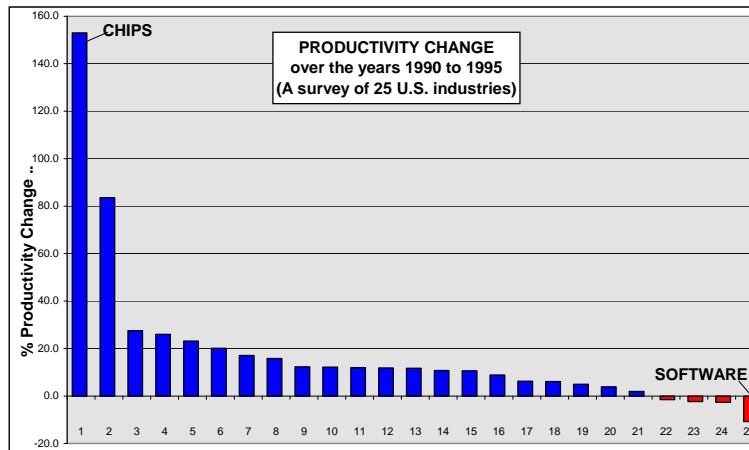
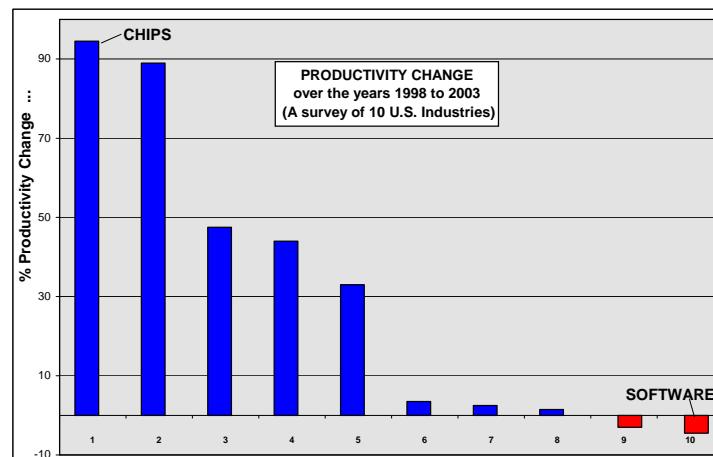


CHART 2. Data From Groth, IEEE Software, November/December 2004 [66]



Recent studies make it clear that this situation hasn't changed. In the year ending 2004 (chart 2), computer chips had the highest productivity growth while software came in with the most negative growth again (red bars). The most recent exposure of the problem is contained in an National Science Foundation RFP, [158]. The underlying cause is the same job protection mentality that occurred on shop floors in the automobile industry in the 1960s, recently putting Detroit into bankruptcy.

But in the computer field it is the approach to software that impedes the potential to dramatically improve productivity using Computer-Aided Design (CAD) tools to build parallel processor software. More importantly, computer chip builders are trying to help solve the software problems with hardware. Precious chip real estate is being wasted on band-aids that only cover up symptoms. As discussed in the next chapter, this is putting Silicon Valley on the same course as Detroit. This book has been written to help put the U.S. computer field back in a sound future direction.

Fair Comparisons Require Representative Performance Measures

One must start with Meaningful Measures Of Performance (MMOPs) when considering such comparisons. To the end users of parallel processors, the major issue is “How much time did it take to run my problem?” Or “How much time did it take to run a day’s worth of typical problems?” And, “How much did it cost?” If time and cost are not measured properly, comparisons may be biased if not distorted, [8].

This is especially true today when some organizations are measuring Watts used to run a computer facility. End users buy or expend Watt-Hours to solve problems. They want to know: “How many watt-hours did it take to run my problem?” or “How many watt hours did it take to run a typical day’s worth of problems?”

Proper measures must account for the fact that some vendors are now building machines with slower memory that takes less watts. To maintain speed, one may need more processors. But as the footprint of a machine gets larger, the distance between processors gets larger, and the time to run the identical test problem can grow exponentially. To obtain meaningful measures, one must compare the *energy expended* to solve the same set of problems.

Finally, different software development environments yield different software architectures on the same machine and correspondingly different speeds. These differences can be huge, and can be illustrated on a single processor. So one must use the same set of problems that fairly demonstrate the comparison desired.

CHAPTER 2. UNDERSTANDING THE PROBLEM

This chapter is concerned with understanding the requirements for a parallel processor software environment. By software environment, we imply both the development and run-time environments - including planned as well as existing hardware designs. Measuring what's best implies comparing different software environments relative to a set of stated requirements. This chapter offers a set of general requirements for both software development and run time environments. Given that a set of requirements is agreed upon, one can then run experiments to compare candidate software environments against those requirements. We note that these experiments must be repeatable by independent parties. Such repeatable experiments form the basis for scientific discovery - the discovery of scientific facts.

We also note that time and cost become major considerations when conducting complex experiments. It is not unusual for a large experiment (or set of experiments) to be conducted in a manner that conserves time and cost. This may result in sets of independent parties participating in a single large experiment, one that is analyzed and documented by all parties. Or by evaluating different production efforts that provide a solid basis for an on-going experiment.

TOP LEVEL REQUIREMENTS

The fact that clock speeds have hit a wall has put a damper on enhancements to existing applications as well as new applications that require faster processors. As described in the prior chapter, this requirement has been addressed by the computer chip manufacturers by putting multiple processors (cores) on a chip and multiple chips on a board. However, the ability to produce run time reduction factors that are commensurate with the number of processors available has been elusive. Large parallel processing applications that are not embarrassingly parallel typically get about a 10% multiplier on the number of processors. In other words, to obtain a speed improvement of a factor of 10 may require on the order of 100 processors. This implies a *processor utilization efficiency* of approximately 10%. This factor, defined carefully in Chapter 6, depends upon the *inherent parallelism* in an application as well as the *software and hardware architecture*. The inherent parallelism properties of an application are described in Chapter 6. Software architecture is introduced in Chapter 7 and described in Chapter 9.

APPLICATION REQUIREMENTS

Embarrassingly parallel applications have inherent parallelism close to 100%. This implies insignificant communications between the separate parts. An example of this is Monte Carlo simulation, where multiple simulations are started with different random number seeds. These applications may be run on separate computers in a cluster environment, or as separate tasks on a server, an easy solution. They are addressed here implicitly, but not of real concern.

The applications of concern have inherent parallelism higher than 50%. Our goal is to achieve processor utilization efficiencies that are above 90% of the inherent parallelism in such applications. In the applications of interest, this typically implies run time multipliers greater than 80% of the number of processors. This translates to a multiplier of 80 or more when using 100 processors as opposed to the current typically good multipliers of 10.

If the technologies presented here can increase the speed of an application running on a single processor by a factor of 5 to 10 over current technologies, the result can be well over an order of magnitude faster on a parallel processor (existing comparisons support these multipliers). The goal of the technology described here from a run-time standpoint is to increase the speed of applications running on a parallel processor by an order of magnitude.

In addition to raw speed requirements, one must be concerned with the application of parallel processing to the simulation of dynamic physical systems. This produces additional requirements for timing and synchronization. These requirements are particularly critical when dealing with discrete event simulations embedded in a real-time system. In these systems, one must be concerned about the unpredictable advancement of the simulation clock as well as the real-time clock. These requirements are addressed in Chapter 6.

Finally, one must achieve high quality while improving productivity. This generally translates to end user productivity which is achieved by designing high quality user interfaces. This often requires complex graphical software interfaces to achieve ease of user understanding and interaction. In addition, the unpredictable nature of timing and synchronization of discrete event simulation (a fast approach to simulation) presents a more challenging requirement for parallel processing. It can be used to build simulations that represent a wide variety of problems, e.g., inherent parallelism, the independence of modules, and communications between modules. This is the application environment that is best suited for testing and comparison.

To summarize, the approach described here addresses large complex applications, typically requiring a team effort to construct. In particular, it addresses applications with a reasonable amount of inherent parallelism so they can run effectively on a parallel processor and meet stringent run-time speed requirements. Examples are real-time planning and control systems used in large manufacturing plants or military operations. These applications require high reliability, rapid enhancement to support new features, and potential for growth of complexity.

There are two basic problems described in the computer literature. First is the need to increase software productivity and run-time speed. Second is the need to simplify software development for parallel machines. These are described below. In addition to these basic needs is the requirement to provide tools for planning and control of complex systems, in real time, as information is fed back to the planners. Examples are: air traffic control; electrical power distribution control; planning large military operations; and top level corporate planning. The application best representative of difficulties in these problems are the Joint Air & Space Operations Centers (AOCs). The solution to that problem applies directly to the other applications as well as the basic needs described below.

The Need To Increase Software Productivity As Well As Run-Time Speed

U.S. software is being outsourced overseas to places like India, China, etc., where programming labor is cheap. Recent articles, e.g., the special issue of the Economist, May 2012, clearly state that software is becoming the most important technology in the world. Productivity in other industries depends upon automation which, in turn, depends upon software. Yet, as described in many references, e.g., [2] thru [4], [6], [19] thru [21], software productivity growth in the U.S. has been more negative than any other industry since the 1980s. We note that, in the last 6 years, except for articles on economics, the technical literature has gone quiet on this topic.

The Need To Simplify Software Development For Parallel Processors

As described in Chapter 1, many articles in EE TIMES and REUTERS describe huge difficulties programmers have using multi-core chips (parallel processors), see Merritt [101], and Sutter [144]. For most large scale parallel processor applications, processor utilization efficiencies are down around 10% or less, i.e., to get a factor of 10 speed increase over a single processor, one needs 100 processors. For applications that have a reasonable degree of inherent parallelism, this should be on the order of 10 processors, not 100. Chip manufacturers, e.g., Intel and AMD are investing many millions of dollars in search of solutions to this problem. But only the subject area experts understand the inherent parallelism in a system. What they need is a CAD system to map their understanding of that inherent parallelism into a software architecture that runs efficiently on a parallel processor. And that is what this book is about.

The Need For Real-Time Planning Tools

There are many planning systems where large sums of money have been invested with little results. The U.S. air traffic control system is an example. What is not understood is that planning requires sufficiently accurate predictions of outcomes of complex decision processes. Predictions are produced as conditioned probability statements, where the conditioning determines the accuracy of prediction. The conditions are represented by models of the system, and the outcomes depend typically on complex physical events whose behavior must be characterized with sufficient detail. In most cases, this can be accomplished through detailed modeling and fast simulations. However, the complexity of the modeling and simulation problem coupled with the requirement to produce results in real-time cannot be accomplished without a breakthrough in software.

This problem is best represented by the planning requirements in Joint/Air Force AOCs, a very complex application that the authors have been working to support for the past 14 years. Prior to that, since 1982, these same authors have been developing a software environment to make efficient use of parallel processors when running large scale simulations. This system, known as VisiSoft has evolved to where it is today, through simulations initially built for the Army, [112], and DARPA, [113] & [114], to demonstrate the solution to the parallel processing problem as well as the software productivity problem. These tests have demonstrated superior results with one exception, the shortcoming of efficiency when using existing compilers and Operating Systems (OSs), e.g., UNIX, Linux or Windows, especially when they are tailored for existing parallel processor chip designs.

These operating systems typically depend upon special compilers or translators that interpret special code or perform tiling, or automatic separation of loops to be run concurrently on separate processors. Users must be concerned about timing and synchronization. As described by the chip designers, these approaches are highly inefficient and difficult to use. For years, Chip designers have been saying that solving this problem requires a dramatic departure from current software development approaches. This has not happened. Instead, modifications have been made to the existing languages, the heart of the problem. In addition, hardware designers have used precious chip space to make up for poor software with approaches, e.g., for hardware cache coherency. This space is much better used to house more memory. Having built the proper solution, it is obvious why current approaches using C-based languages are so far off.

Commercial Market Requirements

Before describing specific requirements for parallel processing solutions, we note our concern with the market for simulation of complex physical systems, a very difficult problem. Although the ensuing discussions apply to other software markets as well, those markets are generally more easily handled by design. Our concern is running large scale simulations, where processors are generally allocated as a group to substantially cut the running time of a single simulation, e.g., from 20 hours to 1 hour or less. In this market, time is of the essence, and parallel processors are purchased specifically to accomplish this goal. The set of assigned processors are typically not shared by other applications.

As in other markets, the benefits gained using a parallel processor must sufficiently outweigh the time and cost to develop and support the software. Otherwise, justification based upon solid economics does not exist. These economic goals will be achieved only if the following requirements are met:

1. The speed multipliers obtained from using a large number of processors must be sufficiently high to justify their cost in time and money (the SPEED measure).
2. The software must be generated and enhanced within a cost and time period that clearly justifies the effort economically (the PRODUCTIVITY measure).

These two requirements are tightly interrelated. Subject area experts typically do not care whether their problem is solved on a single processor or on hundreds of processors. To them, time is of the essence. Without substantial speed multipliers, parallel processing is not justified. If the number of processors and corresponding cost can be reduced by an order of magnitude while simplifying the software, any real market will move to the better solution.

Military Planning Tool Requirements

Visual Software International's client, Prediction Systems, Inc., has been building large scale simulations of complex physical systems since 1982. Current efforts are focused on supporting real-time planning at the operational level for the U.S. Joint/Air Force Air and Space Operations Center. This requires detailed analysis of potential problems that may be encountered when planning large numbers of missions that interact over large areas of the globe, on land, sea, in the air, and in space. Planning cycles using simulation are expected to be on the order of 4 hours, including time for modification of scenarios, running multiple simulations, and analysis.

To perform realistic analyses, one must run large scale simulations covering 1 to 2 hour (or more) scenarios involving hundreds of platforms and the corresponding detailed mission threads in which they participate. This requires models of platform movement, sensor systems, communication systems, command & control decision systems, weapon systems, jammers, etc. of both red and blue forces. Models of these systems must be sufficiently detailed to ensure that realistic worst case variations may be characterized in terms of distributions generated by running 20 to 50 simulations.

A single simulation of this size and complexity may take on the order of 10 to 20 hours to run on a single fast computer. As with most physical systems, there is a high degree of inherent parallelism in the actual system. For example, each platform operates independently except for the required information exchanges between them, with much of the sensor and communications processing being housed on the individual platforms. Many complex Electro-Magnetic (E-M) wave propagation calculations may be done for each individual platform to determine connectivity, with relatively small transfers of information content between equipment on the different platforms.

This form of simulation requires the use of parallel processors to decrease the running time by factors approaching the number of platforms, implying time decreases of 1 to 2 orders of magnitude or more. This makes it possible to run a single simulation in minutes rather than hours. To run even faster, separate simulations may be run in parallel on separate sets of processors. Allowing one hour for scenario modifications and one hour for analysis of results, an operational assessment can be accomplished within four hours, an acceptable time for overall operational planning cycles. With the new technology described here, it is estimated that this may be accomplished using a 32 processor PC that can run faster than a 250 processor High Performance Computer (HPC) using existing technology. The approach to estimating these time differences is described in Chapter 8.

Understanding Real-Time Planning Requirements

The Air and Space Operations Center is typically where theater level planning is done for all airborne operations in multiple 24 hour cycles. These plans must account for ground, sea and space platforms that may interact with those in the air. Large scenarios, e.g., in the Pacific or Mid-East, must provide for hundreds of platforms as they may come and go in a daily cycle. Of particular interest are the plans for many missions within a specified time frame that include platforms with sensors, command & control systems, communications, electronic warfare systems, and weapons.

One of the major requirements is ensuring that platforms that must exchange information can communicate. The Joint Airborne Network Control (JANC) simulation and planning system described here is designed to support that role. JANC can take in Airspace Control Orders and Air Tasking Orders and help to ensure the proper placement and movement of platforms on desired paths (air, land, sea and space). JANC is capable of modeling all of the required platforms and movements in 6 DOF+ as well as the pertinent equipment (antennas, sensors, radios, computers, electronic warfare systems, weapon systems, decision systems), and their environments (electro-magnetic, terrain, foliage, atmospheric, ionospheric, etc.). JANC is designed to help place radio relays and assign flight paths to units to ensure sufficient communications to meet mission requirements.

Plans must account for potential variations and changes that may occur. To do this accurately implies running multiple simulations (e.g., 20 - 50) in a short time period (2 hours) to uncover problems and determine the likely outcomes. This will allow changes to be made in the plans to help ensure the effectiveness of many missions in near simultaneity.

CATEGORIZING PROBLEM TYPES

Embarrassingly Parallel Applications

There are many different types of problems that appear suited to substantial speed improvements using parallel processors. One that always comes up is called “embarrassingly parallel.” These are applications that typically can be broken into separate tasks that can run concurrently since they are almost 100% independent. We will not address applications that are embarrassingly parallel since they may be run on clusters of computers and do not benefit from a single OS parallel processor.

Multiple Tasks

Since the multi-tasking operating systems of the early 1960s, multiple separate tasks have been run concurrently on single processors. These tasks take advantage of I/O wait times to run those portions of tasks that are main memory intensive while waiting for I/O channel responses. Because of the huge difference in time delays to access different layers of memory, particularly mass storage devices, we defer this set of applications to current server environments that make good use of DMA channels. The subset of tasks that do benefit are described below.

Large Single Tasks

Our concern here is with large single tasks that require substantial processor time and have sufficient inherent parallelism in the application to warrant the use of a parallel processor (multi-core) system. This implies that the time spent handling I/O is small compared to main memory processing time. It also implies that the potential speed multipliers that can be gained using a parallel processor are sufficiently high (e.g., above $0.5 \cdot N$, where N is the number of processors). Current speed multipliers may be well under this number ($0.1 \cdot N$ is considered good today), but could be boosted to over $0.5 \cdot N$ (an improvement multiplier of 5 or more) using the integrated software/hardware environment described here. For applications with sufficient inherent parallelism, we are looking to achieve a multiplier of $0.8 \cdot N$ or higher, as shown in the experimental results in Chapter 19.

Software And Simulation

We can also separate parallel processor problem types into software versus simulation. In the case of simulation, one generally uses a simulation clock so that changes or updates occur at a given time based on the simulation clock. Real-time systems, e.g., embedded applications, are typically geared to the real-time clock. Some real-time systems also contain simulations. However, simulation clocks do not affect the timing of software systems unless the simulation clock is tied to the real-time clock, as occurs in real-time training systems. All of these types of system are covered below. Software systems that do not contain simulations are covered in terms of real-time simulations, since the problems they present to a parallel processor and their corresponding solution approaches are contained in that of a simulation.

Types Of Simulations

A large percentage of the current parallel processor applications require simulation, including real-time control and planning systems with embedded models and simulations. These fall into different sub-categories. We will treat them in two categories

- **Mathematical** - The major factor affecting speed is the solution to sets of mathematical equations. Fast solutions to the mathematical problems are found in real-time control systems. These problems are similar if not equivalent to mathematical simulations.
- **Non-mathematical** - The major factor affecting speed involves large complex decision processes or database retrievals. These problems are similar if not equivalent to discrete event simulations. As indicated below in discrete event simulation examples, the difficulties one encounters in software applications are easily covered in discrete event simulation, where on the order of a million *threads* may be active at a single point in time. Special techniques have been developed in discrete event simulation for handling these applications.

Systems Of Differential Equations

Systems of algebraic equations are considered a subset of this category and generally present a much more simple application for a parallel processor. We will use applications that represent a large class of general applications while presenting stress cases for developing software for a parallel processor.

Electrical networks provide good examples of the types of problems one faces when solving systems of differential equations on parallel processors.

- They require solution to a large number of differential equations
- The matrices are sparse
- The time constants vary widely
- The equations are highly nonlinear
- They are used as analogies to solve many other problems

Electrical engineers have developed special techniques for rapid solution on a single processor. These include tabular routines that converge very fast to resolve nonlinear solutions, and optimal sparse matrix inversion routines that eliminate looping as well as effectively all of the parallelism in the system. However, techniques have been identified that allow one to partition a very large matrix into multiple large submatrices using superposition. These techniques provide for nonlinear solutions while making full use of the optimal sparse matrix techniques on separate parallel processors. These techniques are especially useful when running many simulations, e.g., for parametric or sensitivity analysis, where parameters are changed based upon prior solutions. We note that the connectivity of electrical networks is essentially constant. Even though some nodes or links may be removed during the simulation scenario, they may be handled using parameter values of zero or infinity.

In electrical networks, widely varying time constants typically require variable step integration techniques, causing a subset of cells to wait until the others have resolved the solution in preparation for the next time step. This coupled with nonlinear elements will cause the overall solution to slow down to remain in lock-step with the simulation clock.

Systems Of Partial Differential Equations

Many problems require solution to systems of partial differential equations, e.g., fluid or gas flow, molecular structures, etc. In the case of “fine grain” problems described in Chapter 9, one may break the problem into a large number of cells, where each cell is run on a separate processor. In this case, sub-vectors are shared between cells, typically in a 3D array. Using this approach, each cell may solve a set of equations that relate the physics within the cell to the changes at the boundary surfaces. Again, superposition may be used to resolve linear changes. Multiple iterations may be necessary to resolve nonlinear effects within an adjacent cell, slowing down the process to maintain lock-step with the simulation clock.

When the number of cells exceeds the number of processors, a hierarchy of cells may be created where each higher level cell resides on a processor, and each face shares a vector of information that represents all of the sub-cell interfaces with the adjacent face in another cell. This approach has been used parallel processor versions of FORTRAN using a technique called *tiling*, where cells may be grouped as tiles in multiple dimensions, also described in Chapter 9.

Discrete Event Simulation

When problems require complex decision processes that are not easily formulated using a mathematical framework, one must use a *discrete event* approach. In this case, events are scheduled to occur in the future based upon the current state of the system. Examples are simulations of communication and control systems. Simulation of fixed-infrastructure networks is similar to that of an electrical network except that one must implement algorithms that typically receive, process, and transmit messages. The processing can be substantial, involving complex decision algorithms. As the algorithm proceeds, future events may be scheduled that invoke other processes in the system. We note that cell phones fall into the category of fixed infrastructure networks since they are all connected to a fixed base station in the network.

At first, this approach may appear more complex. However, one can more easily represent complex systems using code that resembles (can be identical to) that in the real system. In addition, the simulation clock may jump far ahead if nothing is happening in between, saving much time.

One of the most important factors differentiating discrete event simulation is the state vectors one deals with. In the General Simulation System (GSS), [67], instead of being vectors of numbers, they are *generalized state vectors*, meaning that they hold words or general alphanumeric data as well as numbers. Most importantly, the various sub-state vectors of a simulation are defined in terms of hierarchical data structures, typically containing many levels of hierarchy that can be moved and addressed directly in many ways. Any element of a state vector may be addressed directly by a process that contains English-like statements that engineers and scientists can read and understand without knowledge of programming.

Most important is the separation of data from instructions, so that one can clearly see what processes share what data. This provides the ability to produce architectures using engineering drawings where access to data is defined by connect lines on the drawing. This provides direct visualization of the independence properties of modules, precisely what is needed to produce software architectures that maximize the mapping of inherent parallelism in a system onto the hardware of a parallel processor.

Nonstationary Connectivity

As an example of discrete event simulation, consider the military planning problem, where one may have hundreds of aircraft flying over a large land area, all integrated in a complex scenario. Each aircraft may have sensors, communication systems, weapon systems, etc., that are all modeled in detail. Similarly, there may be many ground vehicles moving on mountainous terrain, some of which may be targets for the aircraft. Although some of the movement may be predetermined by fixed scenarios, most of the movement may occur based upon unfolding events and the decision processes in the communication and control systems.

As the simulation unfolds, sensor and communication systems on a given platform may be connected to different platforms at different times. This is the nonstationary connectivity case. This problem appears to be a significant stress case for parallel processing. In fact it presents an excellent case for understanding the various problems associated with making maximum use of parallel processors.

Processor Utilization Efficiency

Percent processor utilization efficiency can be measured by taking the ratio of the run-times on a parallel processor over that on a single processor and multiplying it by $100/N$, where N is the number of processors used in the parallel case. To obtain 100% processor utilization efficiency, all processors must be working at the efficiency level of a single processor 100% of the time. When some processors are idle - in a wait state while others are working - utilization efficiency drops. Only embarrassingly parallel applications get close to 100% processor utilization efficiency. Even applications that have inherent parallelism at a 90% level will not get close to 90% processor utilization efficiency because of the overhead contained in the current software approaches used for parallel processing.

Non-Linear Problems

The example described above is a highly nonlinear problem, where small changes in a scenario can produce huge changes in the outcomes. To model this accurately requires high resolution models and time scales. To take advantage of huge variations in the required accuracy of time scales - as a function of time - one can use discrete event simulation. Although this technology was developed by Gordon, see [63], [64] and [65], in the early 1960s, it is just starting to be used in engineering. This allows events to be scheduled in future time based upon the current unfolding scenario. In the simulation examples used here, the time between events may vary from microseconds to minutes depending upon where one is in a scenario.

With this type of problem, synchronization of models running on different processors becomes complex. However, the approach described here removes concern for this synchronization from the simulation designer.

Non-Stationary Problems

The example described above is also a highly non-stationary problem. As platforms move on the ground, in the air, and in space, they try to communicate. Communications will depend upon the power level of the signals they receive through their antennas. These power levels will depend directly on various factors, including terrain blockage, antenna gain as a function of orientation, distance, etc. As they move, these factors are changing making their connectivity (ability to communicate) change. Thus a platform that can communicate with a set of platforms now may be communicating with a different set of platforms later. Thus the connectivity matrix is changing with time. If one tries to place platforms on processors that are close to each other based upon the connectivity matrix, then one must be prepared to migrate these platforms during the simulation based upon the changing connectivity.

This problem is prevalent in chemical and biological simulations, where molecules or cells are in motion relative to each other. Again, the effects of one cell upon another depends upon the relative positions of each, and this must be tracked until they are far enough apart.

Meeting The Requirements

To properly represent mission outcomes, one must send the message traffic that determines the success of those mission threads that are critical to completion of a mission. For example, sensor-to-shooter loops typically involve multiple platforms and multiple messages between platforms. When multiple messages are sent during the same time period within a given area using a given part of the spectrum, each receiver trying to receive a message may be hit with power from others. Mutual interference occurs when the desired message cannot be received because of additional power hitting that receiver's antenna, thereby reducing the Signal-to-Noise Ratio (SNR) at the intended receiver below a threshold. This is exacerbated in the presence of jamming, where reception may not occur just because of other noise sources.

Based upon considerable experience, simulations with hundreds of platforms and heavy scenarios will not likely run nearly as fast as real time (they may run 10 times slower than real time depending upon many factors). Requiring more than 20 hours to run a single simulation of a 1 or 2 hour scenario is unacceptable. Furthermore, when using the data from one or more prior simulations to determine changes to the next simulation scenario, the simulations are no longer independent. Thus they cannot be run concurrently as separate independent tasks.

Based upon a number of factors, we have concluded that it will likely take more than 250 HPC processors to adequately support assessments of plans using current approaches, an acceptable range for this application. Using the technology described here, it is expected that this task can be performed using a 32 processor PC. We do not expect current approaches to parallel processing to come close to matching this capability. Furthermore, based upon recent experience, running times on a single processor using Windows or Linux are much longer than what we expect from a single processor using the VisiSoft approach.

UNDERSTANDING CURRENT SOLUTIONS

Prior Approaches

Hardware designers have succeeded in producing parallel and distributed processor computers with theoretical speeds well into the teraflop range. However, the practical use of these machines is extremely limited except on special problems. The inability to use this power is due to the difficulties encountered when trying to translate real world problems with a high degree of inherent parallelism into software that makes effective use of highly parallel machines. This has been described by numerous authors over many years, see for example [8], [13], [103], and [108].

Solutions to the parallel processing problem tend to skip over the software piece of the problem, going from application requirements to hardware architecture. The word *architecture* implies hardware in the parallel processing literature. The words “software architecture” rarely appear, and then the meaning is nebulous. Software design is not much more than an afterthought relative to the size of the hardware design effort.

Most parallel processing hardware vendors have worked to take existing “codes” and split them at run-time to take advantage of a parallel processor. This implies that, if code sequences are invoked in a stationary manner, and memory access is also stationary, then a reasonable solution may be had. But without software architectures that are designed to take maximum advantage of the inherent parallelism of a system, one cannot expect to do much better than a naive model. If one uses tiling as described above, then a run-time system may be able to reorganize the placement of tiles in memory to improve run time speed. But if tiles are not an appropriate representation of the physical system, this approach will not suffice.

However, most applications do not fit this model - they are highly time correlated based upon the unfolding of future nonstationary events. This can only be determined by the software or simulation architect - beforehand. This requires a new approach to building software and simulations as well as a run-time environment that is tied to the development environment.

Requirement For Special Programming Skills

Current approaches to building simulations on parallel processor machines have not satisfied the above requirements except in special cases called *embarrassingly parallel* (e.g., Monte Carlo analyses where each simulation may be run as an independent task). More generally, software development for parallel processors incurs a huge time and cost increase. This is compounded by the fact that the problems requiring large processor power are themselves complex, and best (often only) understood by subject area experts.

For example, a communications engineer trying to design a specific set of algorithms, to implement a very complex set of protocol standards, has difficulty just describing his problem using graphic diagrams with plain English text. To constrain him to describe his problem in an esoteric programming language is difficult. To force him to learn the language of a system programmer is unlikely. To further burden him to describe his problem so that it runs efficiently on a parallel computer makes the current approach intractable.

One is then led to an approach that augments the engineering staff with parallel processor programmers who perform problem translation for the computer. However, it is well accepted in most engineering departments that, when programmers are used to translate an engineer's problem to a computer, problem solution becomes a process whose length increases exponentially with problem complexity. Finally, translation onto a parallel processing machine currently requires very special programming skills that are commensurably scarce and expensive.

This is why engineering departments invest heavily in Computer-Aided Design (CAD) tools that they interface with directly - on their own terms. These CAD tools provide interfaces that are tailored to their problem and automatically generate highly efficient computer code. We believe that this is the only solution to commercialization of parallel computing.

MEASURING VALIDITY

When testing multiple pieces of equipment that interact, one quickly realizes that experiments yielding an identical result are often difficult to repeat. Mechanical systems are examples where tolerances must be considered as part of the design specification. Real time control systems must be designed for large variations. Courses in optimal and stochastic control theory deal directly with this topic. Uncontrolled variations represent a physical phenomenon that must be resolved using sound engineering principles, independent of computers or software. Engineers must negotiate with product managers regarding measures to be used to determine whether or not requirements are met. This implies that various measures of merit, including statistical measures of performance and effectiveness, must be agreed upon - up front.

Validity measures are critical to the design of event driven systems, e.g., real time control and communication systems. Engineers designing such systems typically understand the details of the system requirements better than anyone. They must have detailed knowledge of where and how one can trade off product functionality, cost, reliability, supportability, etc. Without such knowledge, they cannot ensure that the systems they create will produce results that meet validity measures. This is addressed in Chapter 6.

IMPROVING PRODUCTIVITY AS WELL AS CONCURRENCY

The paper by Anselmo and Ledgard, [2], describes the basic properties of software that are required to enhance the ability to produce it, and more importantly, to support future upgrades. Two key properties - *independence* and *understandability* - help to ensure the reusability and scalability of software modules. These same properties provide the basis for designing software architectures that take effective advantage of the inherent parallelism of a system and realize the potential concurrency at run time. The tools used to build that software must ensure that these properties are easily achieved. These tools exist in the CAD environment described here.

Our objective is to minimize the time to build and support a software product while meeting constraints on functional requirements, quality, run-time speed, accuracy of results, and budgets. We leave selection of these constraints to marketing, user groups, and top management. Our goal from a productivity standpoint is to minimize the time to create a new module or modify an existing module. This is the software technology issue, the subject of our focus.

DEALING WITH INCREASING SOFTWARE COMPLEXITY

Recent papers describe the need for an engineering approach to overcome the barriers in dealing with increased software complexity, see [4], [21], [54], [62], [91], and [111]. To accomplish this goal, we must draw on engineering principles that can be used to build software. The principles suggested in the above referenced papers call for techniques such as CAD tools for software, and an approach to modeling the architecture of software systems similar to that used in hardware design.

Such a technology already exists. Having been developed and refined since 1982, it has been described by Anselmo and Ledgard, [2], among others,. This technology was initially developed as the General Simulation System (GSS), used for simulating complex engineering design and building planning tools that require embedded discrete event simulations. This technology is based upon engineering principles and CAD tools used to produce complex electronic circuit designs. Chapter 5 provides an overview of the underlying principles, implementation, and use of that approach. It encompasses a new concept - *software architecture*, and the use of engineering drawings to describe that architecture. This architecture has no relation to flow charts or other symbolic approaches that merely represent code. This technology provides for the design of modular system architectures using measures of independence and understandability.

For large networks, the number of state variables may be in the thousands. Solving worst case design problems involves multiple optimization runs that may require hundreds of simulations each. Each simulation may involve thousands of nonlinear differential equations. Speed and accuracy are the driving forces in designing these simulations. They are also the driving forces that have evolved this CAD development environment.

The underlying problem is to determine: *What is required to improve the way we build software?* To be more specific, what are the factors that affect the *time to build* and *time to run* a large complex piece of software? This sounds like a straight forward problem. Why hasn't it been solved, especially after so many well known people in the field have complained about it?

Conducting experiments to determine the best languages or software development environments is not simple. In fact, it is fraught with potential pitfalls. There are multiple reasons. First, there is a wide range of software applications, from personal to commercial, to industrial, to government and military. Funding agencies from the last two categories have most likely invested the lion's share of the money into investigating approaches to building software. In addition to being prone to politics, neither of these categories contains managers who are *personally pressed* to minimize the time to build or run an application. Their personal success is not on the line - in nearly the same manner - as the owner of a small private software company. The small company must compete for sales of large packaged products - based upon price and performance. Their owners are concerned about personal survival. Government projects are reimbursed based upon cost - by the taxpayers. Management salaries are typically valued in proportion to the size of their budgets, of which the largest corporations win a growing share. Because of their political situation, they are also prone to voicing strong opinions.

Conducting experiments to measure the run-time speed of a reasonably large complex software system is much more simple than measuring productivity, but also fraught with problems. However, given the proper scientific environment, this can be accomplished.

Why Hasn't The Problem Been Solved?

After reviewing the history of programming languages, it is clear that Henry Ledgard's article in the year 2000, *The Emperor with No Clothes*, [89], addresses a major barrier. He quoted W. Edwards Deming (father of quality control) who stated "If you can't measure it, you can't improve it." Ledgard's message: "Without measures from repeatable experiments, software is not a science." This same point was made 10 years earlier by David Parnas, [106]. Considered by most as the father of Computer Science, Parnas was a strong promoter of the College curriculum. However, in later years, Parnas said: "most CS PhDs are not scientists; they neither understand nor apply the methods of experimental science." These views are hard for the software and academic world to accept. Yet both authors are at the top of the list of people knowledgeable in programming language design and have lived in the academic community.

Secondly, measuring productivity is difficult. One must build a sufficiently large and complex system to understand the myriad of factors associated with productivity. Given that one can measure it, one must still compare competing approaches. Performing such an experiment on a sufficiently large scale is prone to many questionable factors. Yet anyone running a software company that is building sufficiently large and complex software is constantly engaged in many such experiments. However, one must be driven by pure economics to fairly conduct trials and take meaningful measurements.

Finally, the VisiSoft CAD environment that provides the dramatic gains in productivity as well as speed is based upon mapping the inherent parallelism in an application system into Independent Modules. This is accomplished by designing software architectures using engineering drawings, and a language that has been carefully designed to support these facilities. This is a huge change from the current approach to building software, and is faced with a correspondingly huge resistance to such change. But the benefits can be shown easily by experiment, measurement, and a fair comparison of results.

Scientific measurements may be difficult to obtain. That does not stop sincere scientists from determining ways to obtain them. An example is the confirmation of Einstein's Theory of Relativity. Einstein waited 15 years before his General Theory of Relativity was confirmed by an experiment that required an eclipse of the sun. Various experiments were set up at those rare time intervals (typically many years apart) and at specific spots on the earth where the bending of light rays around the sun could be photographed. But these failed to materialize multiple times because of cloudy weather. (In those days one could not get beyond a cloud covered atmosphere to look at an eclipse.) Finally the weather cooperated and separate teams of scientists performed the experiment, confirming the accuracy of Einstein's predicted results.

The approach used here is to address this problem in pieces, specifically those that have been part of huge complex software systems, covering commercial data management and Computer-Aided Design with heavy duty graphics, to language translators and parallel processor operating systems. Experiments are defined later in the book that can be conducted at any academic institution seriously looking to understand the problem and seek its solution.

Difficult scientific measurements are performed when a sufficient number of concerned scientists are seeking the truth about a controversial phenomena that they are investigating. Even then, there is ample history of resistance to change (Galileo was jailed for saying the Earth revolved around the Sun).

Why Is There Resistance To Accept The Truth About Technology?

There have been various books written on this issue. Two of them are well cited in the literature. These are Thomas Kuhn's *The Structure of Scientific Revolutions*, [85] and Clayton Christensen's, *The Innovator's Dilemma*, [46]. Both authors cite the same basic factors:

- Not Invented Here - Known as the NIH factor, this is harbored in exclusive private, government, or academic research laboratories where people are getting paid large sums to come up with their own solutions to critical technology problems.
- Job Security - People have spent time learning, using and becoming adept at current technology, whereas the new technology eliminates the need for their special expertise. In cases, the new technology may eliminate a major portion of the profession.
- Financial Competition - Huge investments exist in the current technology, and these are at risk of being wiped out. Large corporations and investment houses have been known to go to Congress to get laws passed or standards imposed to prevent the use of new technology.

The above bullets all support the theme that the approach that currently exists is the best possible approach (don't change horses in mid-stream). These people do not want to hear the truth about the myth - that the current technology must be the greatest (it may have come out of one of the world's foremost research institutions). That is - unless they see a personal gain. In the end, this may be motivated by the handwriting on the wall - for a big potential loss.

LEARNING FROM HISTORY

Those seeking to make contributions to improve a technology generally start by learning the history. As in most fields of technology, the computer field is rich in history. So we will start with a description of as much of that history that we believe to be pertinent. The next chapter is devoted mainly to the development of the computers, i.e., the hardware. The following chapter is devoted to the history of software, and principally the languages that evolved to build applications in software.

In fields of technology, advancements are typically based upon prior history. When those working on advancements read the history, they typically base their new theories on the prior documented results. If the results are not correct, or the presentation not true, the serious researcher may waste much time going down a wrong path. This is why the field of science is based upon repeatable experiments, where independent (unbiased) researchers repeat the experiments to determine if the results are valid.

In general, the history of technology is recorded in the literature. In most fields of science, the literature contains papers that document proposed theories, and more importantly the results of experiments that prove or disprove the correctness of portions if not all of a theory. This is aimed at ensuring that the history presented is true. It is the desire of all serious researchers to avoid the problems encountered, and time wasted, learning from incorrect history. Although many years may go by using the wrong history, it is generally true that the real history is eventually uncovered. On this note, we encourage all readers to be seekers of the truth and contribute to the real history.

Science tries to avoid the potential derailments - driven by money & marketing. But one must be aware that the biggest investments in languages have been put into C, C++, C#, and Java (C-Based languages). Pronouncements of greatness have been made based upon promotion, appeal to style, and popularity - by prior giants, e.g., AT&T and SUN (sometimes referred to as the political approach). This is all at odds with the measures of Hard Science.

But of all the issues discussed above, we must ask the key question: What are we trying to accomplish? Clearly it is to make software easier to build. But what does that mean? How do we measure it? What are the problems people want to solve? How can we define measures of approaches to solutions if we have not clearly defined the problem?

As it turns out, there are many types of problems where software solutions apply. Some are easy to solve. For the easy problems, the solution approach will likely not make much of a difference. What some people consider hard problems, others may consider simple. Also, the Beginner looks at the performance of a Professional and says "That looks simple." The Intermediate looks at the same performance and says "How does he make it look so simple?"

Reading the literature, one often sees snippets of code being applied to a problem. This is often done because the nature of the literature does not permit publication of what may be a huge amount of background material. Without sufficient background, one may not be able to appreciate the context of a given solution. This may be the reason that solutions to problems involving large complex systems are hardly published, except internal to the organization developing the system. The bottom line is that we must take a reasonable look at the myriad of problems that make up the "Software Problem."

CHAPTER 3

THE EVOLUTION OF COMPUTERS

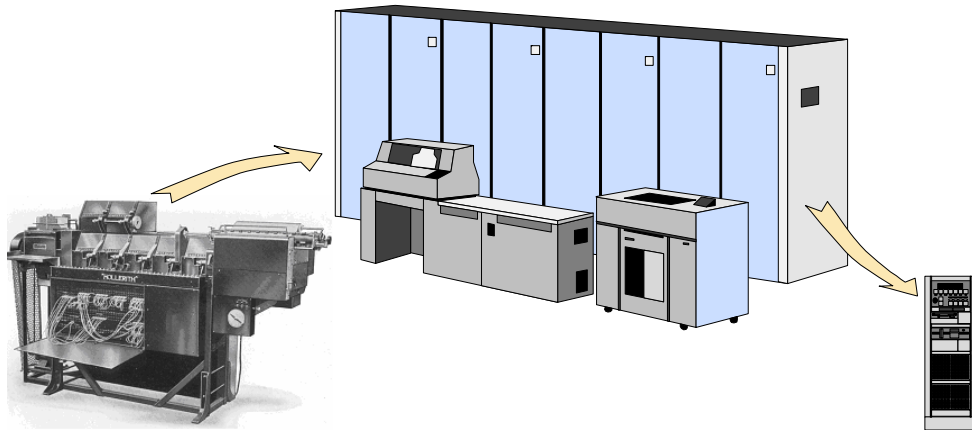


Figure 3-1. From Tabulator to Main Frame to Parallel PC.

PERTINENT HISTORY

Computers have gone through significant changes since their beginnings in 1906, a year most programmers would likely question. It started with the predecessor to the IBM Tabulator using a programmable board (lower left corner in Figure 3-1). The Tabulator was an electro-mechanical device invented and patented in 1889 by Herman Hollerith for the 1890 U.S. Census. It was an essential part of the computing scene for nearly a century. Starting with the 1906 model, “Programmers” had to account for timing and synchronization when wiring the boards that implemented the instructions. Their concerns covered the sequences of electrical signals and timing of mechanical actions, including the resulting delays and race conditions that had to be considered when wiring the boards that implemented the instructions. Although not clearly differentiated at the time, they were really doing logical design. The reason was that, back in those days, computers only stored numbers (and later letters). The instruction sequences were contained on the boards wired by the programmers.

Because the mechanical moving parts in tabulators were slow, it many days to do large calculations. This led to the ENIAC, the first *all-electronic* version, built in the Electrical Engineering Department at the University of Pennsylvania (1943-1945). Designed by Dr. John W. Mauchly and J. Presper Eckert, Jr., its internal memory was used only for storing binary coded numbers. The instructions still used wired programmable boards. Before the ENIAC was completed, Eckert and Mauchly were improving the design, looking to bring the instructions into the same memory as the data. However, at that time, memory was quite limited and the search was on to find ways to increase memory.

The Von Neumann Architecture - A Programming Breakthrough

As electronics replaced mechanical devices and new devices were conceived to expand memory, John Von Neumann, from the Institute for Advanced Studies (IAS) at Princeton University used the new memory to develop an instruction set that would support his work on solving the shock wave problems for nuclear bomb calculations. His instruction set, combined with Eckert and Mauchly's concepts on computer architecture, moved the instructions from wired boards into the same memory that previously stored only data.

This important breakthrough in computer programming was described in a paper by von Neumann, see [148], that contained a plan for the EDVAC, a follow up to the ENIAC,. The instruction set, to be stored in memory along with the data, became known as the Instruction Set Architecture (ISA). The ISA was driven by von Neumann's application requirements to solve sets of differential equations using discrete time difference equations and binary numbers. Implementation of this theoretical paper dramatically simplified computer programming. The concept was implemented to a very small degree in a later version of the ENIAC, but in that modification the instructions were still fixed, implemented by switches instead of cables. As a result, they were not necessarily sequenced in order, causing reliability problems due to timing.

The first computer design was implemented in the MANIAC, built at the Princeton IAS. It was aimed at supporting continuation of the Manhattan project to develop the Hydrogen bomb at Los Alamos, NM. This new paradigm forced programs to follow a sequence of instructions, resulting in the *independence* of sequential operations. Timing and synchronization within a given instruction were left to the logic designers of the machine. Timing between instructions was no longer a programmer concern. This inherent independence property provided a great programming simplification, ushering in dramatic increases in software size and complexity.

One of the first fully programmable "von Neumann architecture" computers was the PENNSTAC built at the Pennsylvania State University (1953-1957). The PENNSTAC was programmed by entering instructions into its large drum memory (donated by IBM) using a flexowriter (a souped-up teletype) and printing them out for review. Programs could also be punched on paper tape for storage and subsequently modified off-line.

The instruction set on the PENNSTAC was reasonably sophisticated for its time. Instructions could modify themselves - and the program - on the fly. A course in logic design, taken by author Cave, required writing programs in 1s and 0s (on standard binary coding sheets).

Mnemonic code translators quickly appeared so that instructions were designated using 3 letter labels, and memory locations were specified in decimal. Data could also be specified in terms of decimal numbers or letters. Large numbers had to be handled by the programmer using a pair of numbers. The mantissa was used so that the first bit was always in the left most position to maintain accuracy. The number of shift-left operations was tracked using a scale factor, a form of binary exponent.

Programs containing both data and instructions could be loaded from paper tape into the machine to run. Additional sets of data could be loaded through paper tape once the program was loaded and running. This was a major breakthrough in programming, allowing relatively large complex programs (for that period) to be built and stored for future use, including the use of subroutines that could be patched into multiple programs. Programmers kept their routines on paper tape - wrapped in rubber bands and hung on nails in a board - next to their desks.

THE NEW ENVIRONMENT

Figure 3-2 illustrates the parallelization of platforms that has evolved since the early days of large scale computing. Most computers today may be viewed as parallel platforms communicating with each other. Each platform is managed by its own OS with tasks that are independent of those on the other platforms. Except for minor communications between some of them, they typically share files. From manufacturing to retail, businesses depend upon platforms tied via networks. People at the cash register do not want to wait more than a few seconds to complete a sales transaction. Time is money. This requires fast communications.

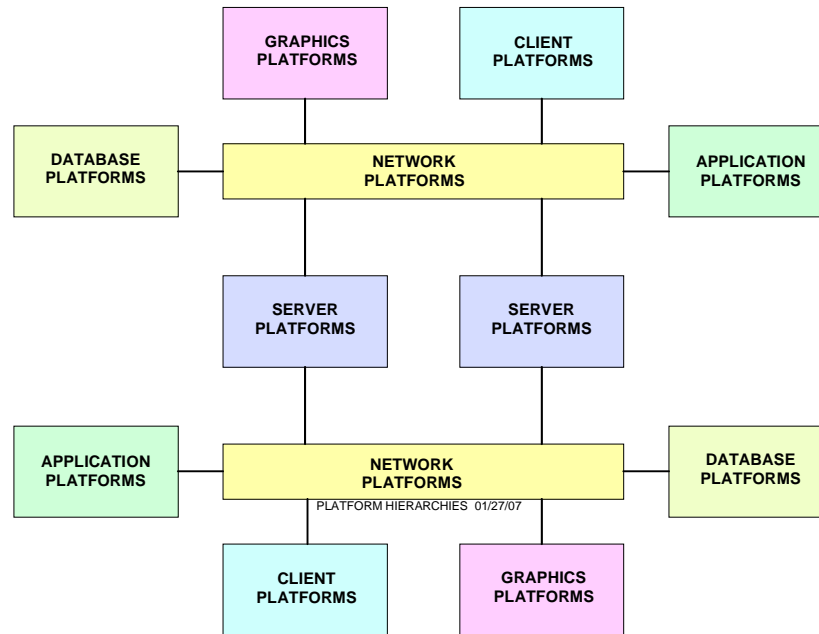


Figure 3-2. The parallelization of platforms.

GETTING CLOSE TO THE MACHINE

Looking behind each platform in Figure 3-2 we find a “machine”. There is growing agreement in the literature about where the “machines” are headed. Most of the recent literature is looking at multi-core chips, because that is what is being manufactured. The new chips offer multiple processors (cores), albeit each may be slower than a fast single processor. As an example, these chips may be grouped onto boards with 32 processors. An example is depicted in Figure 3-3. In the case of High Performance Computers (HPCs), Boards are then grouped into trays, trays into racks, and racks interconnected. This is different from the multiple server environment because all processors are managed using the same OS. Writing a single task to run under a Single OS (SOS) and effectively use multiple processors is a new problem to this market.

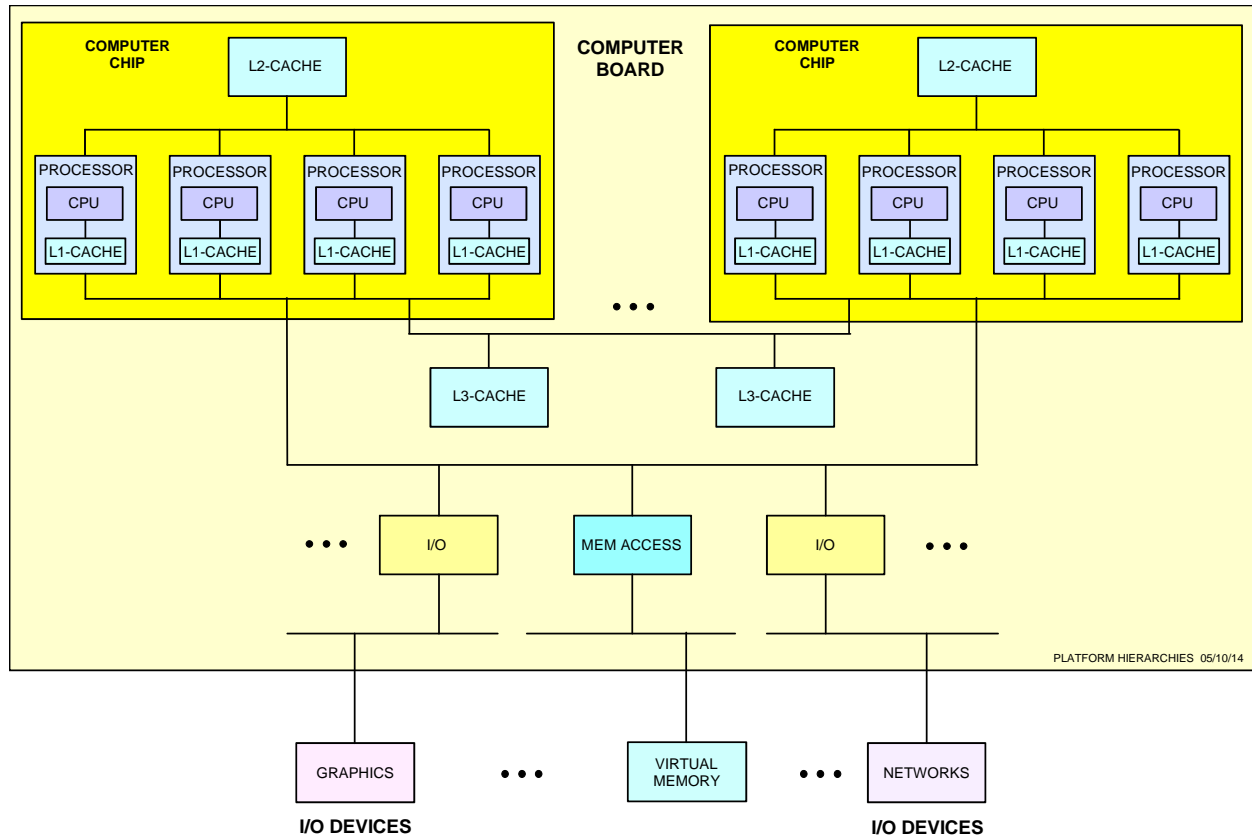


Figure 3-3. A parallel processor computer board.

Then there is the engineering market where hundreds of processors have been put together under a SOS parallel processor to run large scale simulations. This is somewhat more difficult. One has to maintain synchronization with a *simulation clock*, a much more tedious problem than just synchronizing with a real-time clock. Unless synchronization with the simulation clock is maintained, simulation results quickly become chaotic - as well as invalid, see [114]. We also note that, in the case of real-time simulations, the simulation clock must be synchronized with the real-time clock.

Many authors have documented experiments using large numbers of processors to run a single large scale simulation. This provides a significantly different viewpoint from that of managing two, four, or even eight processors. It is a viewpoint that magnifies the difficulties in approach. Much test data exists for various schemes designed to solve this problem. The results have been counter-intuitive as illustrated in Figure 3-4. In most cases, speed multipliers fall off quickly if not going fractional (faster to run on a single processor). This is because of the level of complexity that must be dealt with, and the totally different space of solutions one must consider. The result is that most parallel processor suppliers are no longer in that business. Except for special applications, building software to take advantage of their hardware has been much too difficult.

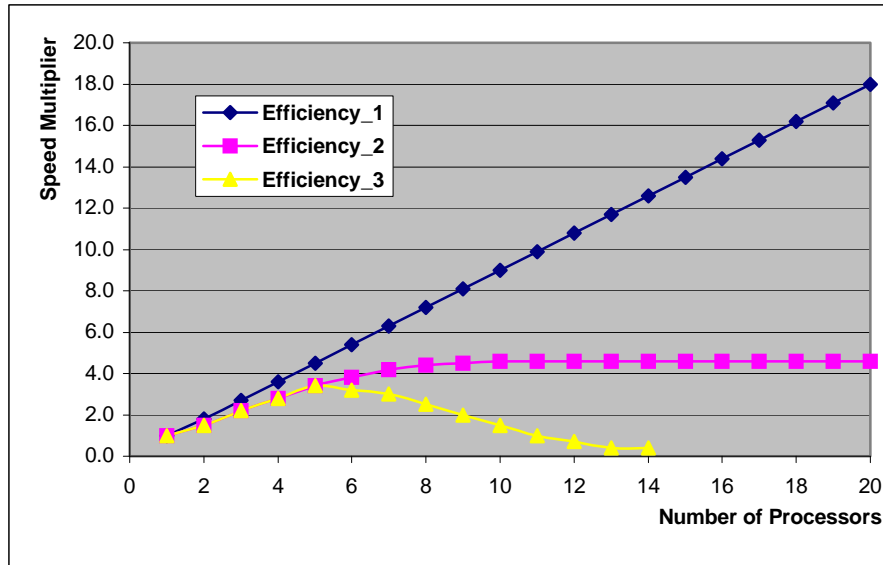


Figure 3-4. Efficiencies of different software approaches to SOS parallel processors.

What is more important is that programming approaches that make software slower on single processor platforms are going to make things *much slower much faster* on a SOS parallel processor (an exponential form of Wirth's Law). Piling layers of abstraction on top of the existing layers moves us further from the machine faster.

Figure 3-2 embodies the embarrassingly parallel case. Tasks are restricted to a single processor. Programs follow the sequential nature of the Von Neumann architecture. Transfers between processors are simplified by the use of Inter-Processor (IP) communication protocols. In the SOS parallel processor of Figure 3-3, one is confronted with Efficiency_3 in Figure 3-4. We will explain this problem and its solution below.

Before addressing the problem of SOS parallel processors, we must understand the next layer down, the basic processor itself. If we are not close to the single processor, then with N processors we are going to be much further removed. We must map software into one processor effectively, assuming that we are going to multiply it by hundreds. If we are not thinking in terms of using hundreds, we are not working in a space that will achieve an effective solution.

Understanding The Single Processor

Almost all R&D money spent on parallel processing to date has been on hardware, with software an afterthought. Going back to the Holland Machine in 1958, [74], hardware designers have tried to produce new architectures that improve on Von Neumann's original design. Instead, simplicity of programming has caused this basic design to evolve over the past 50 years to the latest chips as illustrated in Figure 3-5. The most significant architectural changes are the move from words to bytes (IBM 360), and the separation of data memory from instruction memory (PC on a chip). Although other benefits were realized, the driving force behind both changes was speed.

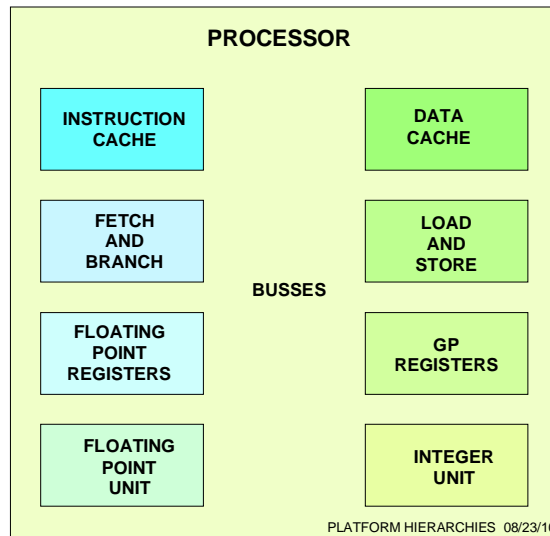


Figure 3-5. Illustration of the “new” Von Neumann architecture.

Behind most every platform today is a processor similar to that illustrated in Figure 3-5. Speed is determined by a number of factors. Referring to Figure 3-2, applications consist of functions that end users want to perform. Users generally work interactively on client and graphics platforms. In the client server market, application platforms process client input data using databases and return outputs to client and graphic platforms as well as updating databases. Most factors affecting speed are associated with transferring data (communications).

In band-limited systems (computers and busses), minimizing data transfer times implies minimizing the data transferred. In Figure 3-2, copies of databases are common, and transactions are identified by simple codes and numbers packed into records for transmission. This implies designing data structures that minimize the data transfers required to perform the desired functions. Transferring programs (instructions) is avoided when speed is important.

Using a single processor, multiple tasks may run in *virtual* concurrency, with the OS managing utilization of the processor based upon task priorities and resources requested from each task (e.g., memory, I/O devices, etc.). The OS swaps blocks of instructions in and out of instruction cache and pages data in and out of data cache. Swapping and paging is done based upon where things reside (the *context*) in the memory hierarchy (cache, RAM, disk, etc.).

But one must put these delays and time-frames into perspective. In the case of interactive systems, response times of milliseconds (if not seconds) are typically sufficient. When looking at the difference between memory boundary crossing delays, differences range from nanoseconds to microseconds. To appreciate the meaning of these delays, one must gain a perspective on the difference between the effects of milliseconds, microseconds, and nanoseconds at a particular point of interest in the processing cycle.

BEYOND THE SINGLE PLATFORM

Definitions

It is clear from the definitions provided below that the hardware architecture for a parallel processor must be much different from that of a server.

Server Systems - These typically support large telecommunications requirements for transaction processing and large database management applications. A large server system must interface with huge I/O facilities, including networks of workstations and big disk management. Servers are composed of large numbers of processors, where each processor is typically running multiple tasks with fat communication channels to fast I/O, including teleprocessing channels and big disk facilities. Applications include large commercial data processing, huge database management including query, and remote teleprocessing and cloud type applications. It can also support embarrassingly parallel applications defined below.

Parallel Processors - These are required to support true parallel processing applications as opposed to embarrassingly parallel applications (see below). Parallel processor applications have substantial inherent parallelism, i.e., elements that operate independently, that can be put into separate software modules. They require a large number of processors running in parallel with these independent modules to meet the time constraints for a single task. They require limited one-way I/O (typically output after initialization). They typically require intensive internal processing of mathematical systems or decision processes that are processed in parallel. Examples of parallel processor applications are EM wave simulation, meteorological simulation, and fluid dynamic simulations (e.g., fluid flow through multiple container surfaces; moving particle physics; and dynamic biological, and chemical particle interactions).

Embarrassingly Parallel Applications - These may be broken into multiple separate tasks running on separate processors. Once they start to run, they need little if any communication between processors. They can be run as fast on a server, or a cluster of PCs, as on a true parallel processor. Scientific applications, e.g., Monte Carlo simulation and fast approaches to large scale Linear Programming (LP) are embarrassingly parallel. These applications are poor examples of the requirements for Parallel Processors. However, they are often used by those pushing fast server or cluster environments.

At this point we must differentiate between Input/Output (I/O) transfers, i.e., memory to I/O devices, and memory-to-memory transfers that occur in a typical server environment as shown in Figure 3-6. Figure 3-5 illustrates the processor CPU and on-chip memory only. Additional devices, including further hierarchies of memory as illustrated in Figure 3-3, may surround the single platform chip. Large blocks of instructions that use local memory transfers can be processed fast compared to a single I/O instruction. In these cases, the OS can determine when swapping and paging among tasks are effective. Because I/O transfer delays are easily isolated with good software architectural approaches (described later), our interest here is the speed of internal processing in a large single task that would run on parallel processors.

GENERAL PARALLEL PROCESSOR FACILITY

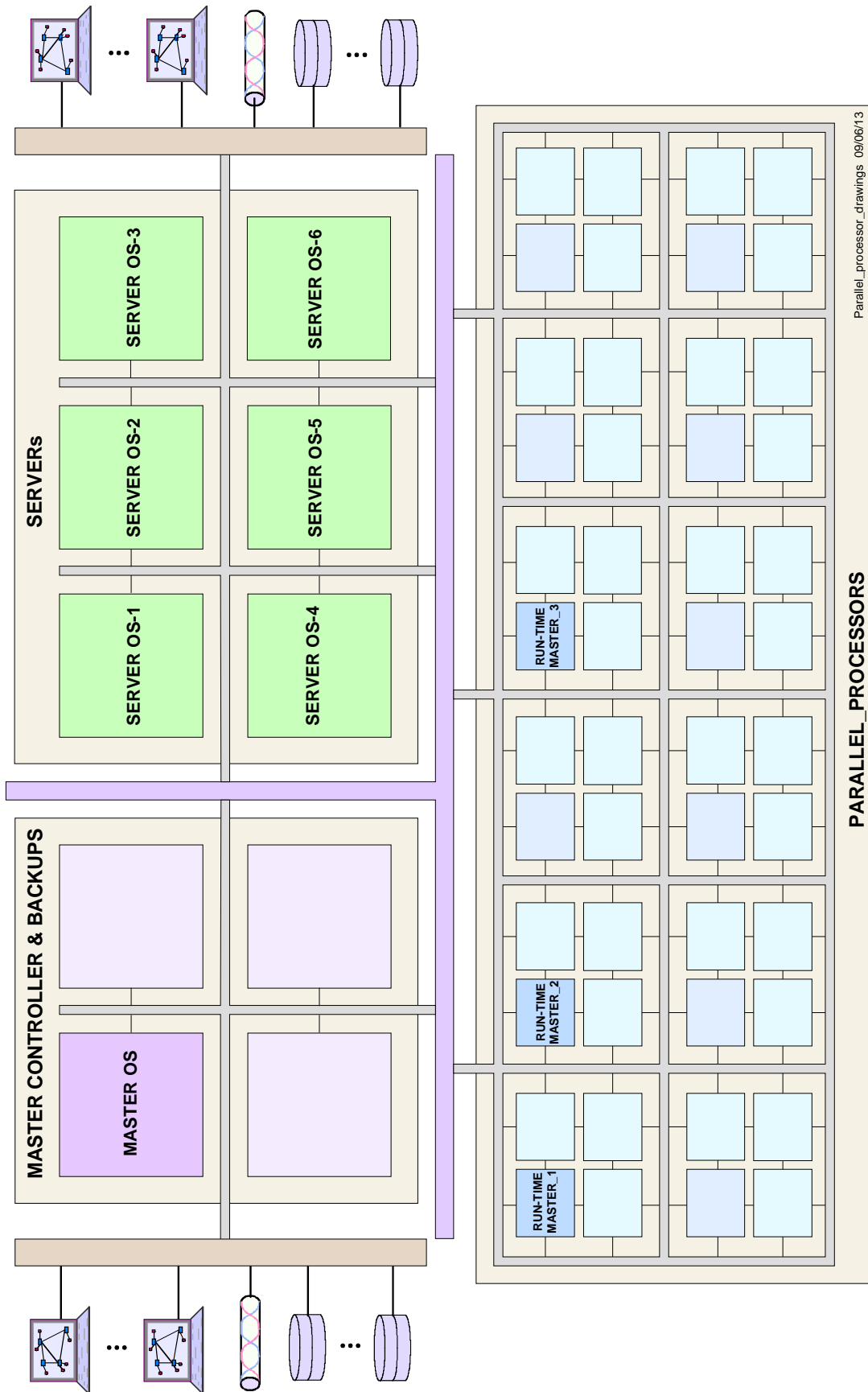


Figure 3-6. Server environment with parallel processors.

As a single task grows in size, swapping and paging may occur when it is the only task running on a single processor, with or without I/O transfers. It is not unusual for large tasks to use huge data spaces that require many gigabytes of storage. Transfers across memory hierarchy boundaries using multiple communication busses may incur relatively significant increases in delay. These are caused by speed differences in memory types as well as transfer delays on the interchange busses.

In the case of a single task on a single processor, transfers across memory boundaries can have a significant impact on speed. To maximize speed, the OS attempts to map out memory in a way that minimizes swapping and paging. This implies predicting what blocks of memory will be used beyond the current block of instructions. However, programs containing complex sets of algorithms that deal with large blocks of data are only understood sufficiently by the software designer - not the OS designer.

When dealing with large discrete event simulations of the type required by the authors, the use of memory is somewhat unpredictable. This is due to the non-stationary nature of the scenarios that depend upon interactions between processes. This problem and its solution will be explained further in Chapter 14. However, with current approaches to programming, it is doubtful that even a good software designer can make such predictions. Expecting such predictions from the OS is folly. Depending upon stochastic time constants, migration may trail far behind the applicable statistics.

When considering applications that may run on parallel processors attached to a server environment as shown in Figure 3-6, one must consider an OS design that best supports the different applications. For example, when using a large number of parallel processors, it is best to exclude direct connections to I/O devices from the parallel processor environment, and deal with these devices through a server. This allows the server to manage moves of large blocks of memory to and from these devices without slowing the parallel processing task. If the software architecture on the parallel processor can identify one-way transfers, then reading files for initialization and writing files for output can be serviced without slowing down the heavy processing burden. This is also true when producing significant data output to a graphics workstation.

Figure 3-6 illustrates a scenario where three parallel processing tasks are running concurrently on different sets of processors. These tasks may be managed by different servers that interface with different I/O devices. We note that the number of servers and parallel processors in the figure may be small compared to some actual environments. However, we expect that the parallel processor part of that configuration will fit in a tightly coupled set of PC boxes (if not a box the size of a PC) within a year or so, implying the Personal Parallel Computer (AKA the Parallel PC) will solve most real parallel processing problems.

Instruction Set Architectures

This chapter started with a description of the first stored program computer that opened the door to the huge world of software. Although von Neumann generally left logic design to the engineers, he was given the credit (rightly so) for the architecture that, still today, is known as the von Neumann architecture. What he contributed was the definition of the instructions to be implemented by the hardware, an approach that became known as the Instruction Set Architecture (ISA). Although computers are defined by their instruction set architectures, the hardware implementation can vary to maximize speed, minimize power dissipation, and minimize cost.

Going back to the 1980s, Silicon Graphics Inc. (SGI) was created based upon a similar concept applied to graphics and visualization. The original language, SGI-GL, was developed by James Clark and his team while he was at Stanford. That team fostered the start of SGI. Their Geometry Engine was the hardware that supported the higher level graphical instructions (the ISA). In this case, the graphical ISA defined the requirements for the logic and hardware design that implemented complex 3D transformations and data pipes to speed the production of dynamic graphical images on a computer screen.

While Clark was Chief Technology Officer at SGI, the SGI team learned more about the requirements for graphical software, and SGI-GL was replaced by what is known as Open-GL. Open-GL expanded the language facilities while greatly improving and simplifying the language interface from a user-functional standpoint. Open-GL is now the standard ISA for fast graphical computation and display, and is implemented differently by various graphics chip manufactures.

Application Space Architectures

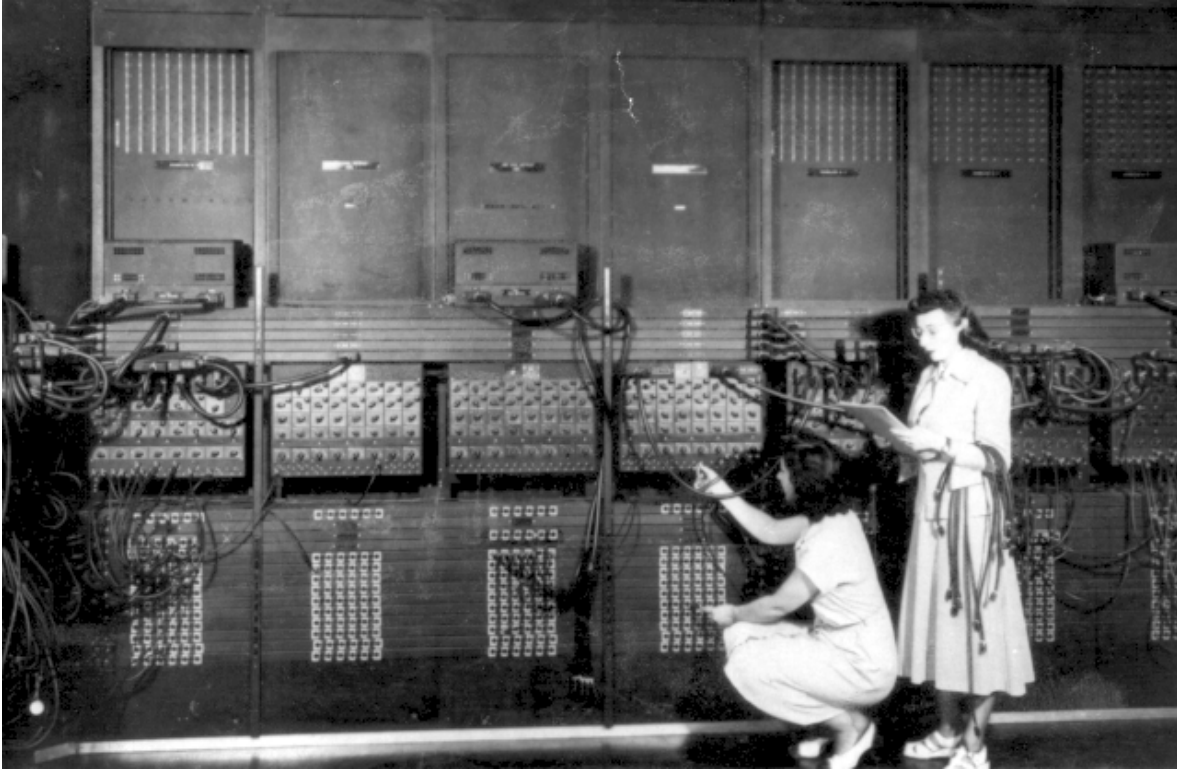
Why can't large complex software systems be decomposed into independent modules that run concurrently on relatively standard processors that share memory? The answer is "They can!" But, as stated by top engineers referenced above, this requires acceptance of a disruptive technology, i.e., a totally new approach to designing software. Without an Application Space Architecture (ASA) - the equivalent of von Neumann's ISA for parallel machines - new approaches to hardware continue to ignore history.

The parallel processor applications enumerated above all have different software design requirements. The spaces required to minimize complexity of their algorithmic solutions are all different. Representing these problems requires hierarchical data spaces and discrete event spaces - as well as continuous and discrete time spaces - all in the same application. To support this translation requires an extension of mathematics that helps one to conceive the design of these spaces and corresponding decision processes. The application software must also invoke synchronization facilities for sharing temporally independent data spaces that only application area experts will understand. Software development must be easy for these experts. Given a language that implements these facilities, one is on the road to determining the requirements for parallel processor hardware design.

The ASA presented here covers a broad set of applications that can be tailored to a well-defined hardware architecture, while leaving the door open to those who can afford hardware architectures tailored to specific applications.

CHAPTER 4

THE EVOLUTION OF SOFTWARE



Authors' note: *In presenting the history below, we emphasize two important ideas in software. First is “independence”, by which we mean the ability to make modifications to a software module without affecting other modules. Second is “understandability”, by which we mean the ease with which a programmer can read and change a portion of software written by another.*

IN THE BEGINNING ... (circa 1955 - 1975) - *DRAMATIC SPEED IMPROVEMENTS!*

Back in the old days we wired boards. After the big breakthrough, we wrote code in ones and zeros - using binary coding sheets. To be a programmer, one had to understand the machine. Programming required knowing the memory layout, program counter, registers, arithmetic instructions, control instructions, I/O instructions, etc. Writing a simple program was not easy. One had to define what was in each memory location and clear registers before they were used. Figure 4-1 illustrates the format of a program. This example is for a fictitious, but simple, single address machine with an A register. For example, OP Code 1000 cleared the A register to zero and then added the contents of the specified memory address, e.g., 13, into A (in this example, location 13 contains the value 25).

Notice “the separation of data from instructions”. The data could be put anywhere. The instructions had to follow in sequence, unless a transfer (GOTO) was used.

MEM LOC	OP CODE	MEMORY ADDRESS	COMMENTS
1	1000	00001101	CLEAR AND ADD [13] TO A
2	1001	00001110	ADD [14] TO A
3	0010	00001111	READ TAPE INTO 15
4	1010	00010000	SUBTRACT [16] FROM A
5	0111	00001110	STORE A IN 14
6	1110	00001011	TRANSFER TO 11 IF A IS NEGATIVE
7	1100	00000010	TRANSFER TO 2 IF A IS POSITIVE
8	1001	00001111	ADD [15] TO A
9	0011	00000000	PRINT A
10	1111		STOP
11	0011	00001111	PRINT [15]
12	1111		STOP
13	000000011001		25
14	000000011010		26
15	000000000000		0
16	000000110010		50

Figure 4-1. A computer program written in binary.

We note that, during this era, use of the binary number system started to become wide spread. Every binary string could be treated as a letter or string of letters, even though it represented a number. Claude Shannon used this in his “mathematical theory of communications”, [130], where he defines measures of error. In a later chapter, we provide an extension of mathematics, one that includes words, and IF ... THEN ... ELSE conditions.

Working in the binary number system was not the most difficult problem. Modifying the program was the real challenge. Consider that one wanted to put a few additional instructions into this program starting at memory location 8. Then, all entries from there down would get new memory addresses, implying that every reference to them must be changed, a real mess! Even if one was clever enough to insert a GOTO to some higher location with new instructions, one still had to move the old instruction in 8 to the new location, and put another GOTO at the end of the new sequence (to get back). Time spent debugging these changes and random jumps was immense. The important lesson here is that all lines of code were dependent upon the sequence, and thus each other. This *lack of independence made change very difficult*.

The Properties of Understandability and Independence

Back in those days, computers were used by engineers and mathematicians trying to solve real problems. Programming was not their interest. Their concern was how long it took to get a program running correctly, and then how long it took to run. It was not unusual for those people to complain about the stupidity of the machine. But for large problems, it was still much faster than using a hand calculator. It didn't take much time for these end users to start writing translators to make it easier to write programs. This implied making them more readable, and therefore more *understandable*. The first simplification was the use of *mnemonic code generators*, using names for Op Codes (e.g., ADD) and using decimal numbers for addresses (e.g., 65 instead of 01000001).

THE JUMP TO ASSEMBLY LANGUAGE

Indirect Addressing

Mnemonic code generators quickly evolved into the next productivity improvement: the *assembler*. Each machine had its own assembler that did much more than mnemonic code translators. The first assemblers provided indirect addressing, allowing the use of symbols instead of specific numeric addresses. With indirect addressing, actual addresses were determined by the assembler and loader at run-time. This was a major improvement - even for smaller programs - where labels were referenced instead of actual memory addresses, see Figure 4-2, allowing one to insert lines of code without changing the referenced addresses.

LABEL	OP CODE	MEMORY ADDRESS	COMMENTS
RESTART	CLA	X1	A = X1
	ADD	X2	A = A + X2
	RDT	Y1	READ TAPE INTO Y1
	SUB	Y2	A = A - Y2
	STO	X2	X2 = A
	TRN	END2	TRANSFER TO END2 IF A IS NEGATIVE
	TRU	RESTART	TRANSFER TO RESTART IF A IS POSITIVE
	ADD	Y1	A = A + Y1
	PRN	A	PRINT A
	STP		STOP
END2	PRN	Y1	PRINT Y1
	STP		STOP
X1	25		
X2	26		
Y1	0		
Y2	50		

Figure 4-2. A computer program written in early assembly language.

Subroutines And Relative Addressing

As programs grew in size, it became desirable to split them up into separate sections. This led to the separation of programs into subroutines, with provisions in the assembly language for calling to them by name and returning automatically. Programmers also had to specify a starting (MAIN) routine. This required relative addressing where the addresses within a routine were relative to the first address in that routine. Multiple routines were assembled and linked together with all real memory addresses determined by the assembler.

Linking And Loading

Relative addressing allowed programmers to add or change one or more routines and then reassemble them with the others. Using a program loader to run the program, one could typically specify the starting memory address. The program was then loaded into memory. As many subroutines became more reusable in different applications, programs grew even larger, leading to another problem. All routines belonging to a program had to go through the assembler to link and load. As main memory became larger, making a simple change to a single routine still required the whole program to be reassembled and loaded.

This led to the separation of functions, namely language translation, linking subroutines together replacing the relative addresses, and loading the program into memory ready to run. These functions were separated off from the assembler into a Linker and a Loader.

A single subroutine could be assembled *independently* of the rest into a binary 'object' module. The linker would link the object modules (binary routines) produced by the assembler into a linked object module containing all of the routines where all addresses were resolved relative to the top routine of the linked module (MAIN). When the program was totally linked, the loader would load the program with real memory location addresses starting at a pre-assigned starting memory location.

As linkers became more sophisticated, they could link object modules that had already been linked with those not yet linked. This provided for libraries of linked object modules that were already assembled and partially linked into different programs relative to a top routine. This allowed for libraries to be shared as linked object modules so that the source code was controlled by a single developer. This allowed application programmers to use complex library modules that were well documented, well tested and reliable. These object modules resided as independent entities in library pools that were scanned during a subsequent link stage.

Floating Point Arithmetic

In these early days, programmers had to provide for overflows and underflows when using multiply and divide instructions. This required storing a *scale factor* along with the number to track the position of the decimal point. This corresponded to the number of shifts - left or right - required to maintain a binary exponent that ensured the mantissa was stored with the *most significant bit* in the highest bit position. As transistors came into use and logical designers started using Computer-Aided Design (CAD) systems, instruction sets became much more sophisticated. For example, real numbers could be specified directly and manipulated with automatic handling of exponents as well as providing maximum accuracy. For scientific computation, this provided major improvements in productivity.

A STRONG DRIVE FOR IMPROVED PRODUCTIVITY

With indirect and relative addressing, new code could be added anywhere using labels for reference and relative addressing for subroutines. Productivity shot way up. Having written many programs without these assembler facilities, one understood how such user-oriented language changes could provide substantial reductions in the time required to produce reliable programs.

In these early days, most of the people writing programs were end users, typically engineers, mathematicians or accountants who were experts in an application. To them, computers were a tool to produce solutions. They were driven by their own productivity, i.e., *the time it took to solve their real problem*. They wanted to minimize their time spent writing programs. Anything they could do to reduce this time was sought and used. Productivity was clearly measurable based upon one's own experience. This drove rapid improvements in the ability to write, debug, and run programs. These improvements were quickly shared. This created the ability, and therefore desire, to solve even bigger and more difficult problems.

Overlays

As more memory became available, computer programs quickly grew in size to meet application speed demands. Disk and drum memories became sufficiently reliable so that larger programs could be stored on these external devices for days. Programs could be loaded rapidly into main memory when they were needed. However, internal memory was expensive and scarce. As program sizes grew, they quickly exceeded the internal memory of the machine. The ability to overlay modules that were no longer needed soon became a necessity. This was particularly true for handling sequential I/O devices such as tape file input and output routines. Large input files were read, databases processed, and large output files written.

This led to the desire to overlay memory containing routines that were no longer being used, with those routines that were needed but sitting out on disk. This required changes in the design of instructions so they could use pointers to a base address used by relative addresses at run-time. When routines were brought into memory, the base address was assigned the starting location of a real memory address that was followed by sufficient free memory to store the overlay. The base address was loaded into a relative address register so that all instructions or data referenced within that overlay were relative to its base address.

Run-Time Memory Management

As overlays became popular, application programmers had to lay out the patchwork of where routines were best mapped into memory. This was solved using *memory managers* implying that the real memory starting addresses of the overlays remained undecided until they were loaded into memory. It was still up to the application programmer to design the overlays accounting for the splitting and reuse of functions and corresponding memory sizes. This provided for *spatial independence* of overlays, relative to where and when they were mapped into main memory, making the memory management much easier while programs ran much faster. Overlay programs became a general requirement that led to the Operating System (OS). Other functions, e.g., identification and management of files on disk, and running a sequence of tasks as part of a job stream created the need for OS level functions.

A growing list of library routines created the next problem, that of duplicate names. To this day, the problem of duplicate library names, amplified by flat file object libraries and very simple library managers and linkers, plagues a growing part of the programming world. This has led to a lot of band-aids in programming languages to cover up problems that are best solved at the software environment level. These solutions are addressed in subsequent chapters.

THE GREAT LANGUAGE PRODUCTIVITY BREAKTHROUGHS

Computers were a scarce commodity in the early days and people lined up to use them. Anyone wasting precious machine time to get programs running was marked. Then came the first compilers (FORTRAN and COBOL circa 1960). This split the programmers into two groups: the assembly language generalists, and those working to get answers to their application problems. Having written a significant amount of scientific code in binary, and then using a relocatable assembler, it was obvious how languages could help solve problems much faster.

Back then, no one argued the use of binary coding as being more efficient than assembly language programming. Too many people were familiar with both, and how run-time speed was greatly influenced by managing memory. Today, few people understand the tremendous shortcomings of assembly language. They have little experience with the difficulties in the layout of and access to large data hierarchies that dramatically improve speed. They are prone to believe that lower level languages are more efficient. When compared to a good high level language, this is easily proven to be totally false.

With the advent of the mainframe, one had to go to the programming shop to get large programs written. This did not sit well with computer engineers developing CAD systems and complex scientific programs. The programming shop also had priority over machine time - day and night. Engineers could only get time at night, typically implying one shot in 24 hours. That meant putting in 3 or more runs each night. Using compilers was essential in this limited environment. Yet the programming group refused to write in FORTRAN or COBOL, saying “efficient” code could only be written in assembler!

This logic implied that writing in binary should be even more efficient. Even after the belief in assembly language proved totally false, the assembler programmers refused to give it up. What was their motivation? To get programs written by the programming group, one had to allocate money - in the form of time cards. *They were paid by the hour.* When the number of jobs started to dwindle, they spread their time on each job. *Improving productivity was not of interest.*

FORTRAN - A Big Jump In Engineering Productivity

The desire to make the programming job easier led to a major step toward making the machine do more of the work of understanding the language of humans. People writing programs to solve large sets of mathematical equations were the first to invent a more *understandable language* and corresponding translator - the FORMula TRANslator (FORTRAN). FORTRAN shifted the burden of translation from the person writing the program onto the compiler writer. Productivity went way up because of a number of factors.

- One person could understand much more easily what another person wrote (or what that same person wrote a year ago). Equations were written directly - as in a math text. Conditional statements were easily understood. This allowed a large program to be constructed with a team effort. It also allowed completion of an effort and reuse of code without the original author.

- Many errors endemic to assembly language disappeared. Probably the most common was scribbling on instructions (and immediately the rest of memory.) FORTRAN took away the Von Neumann facility of being able to write instructions that modified themselves. This was backed up by the OS.
- More and more smarts were built into the translation process as people compared what it took to be more productive. These included improved syntax, various forms of error checking and prevention, run time messages, etc.

It is interesting to note that many programmers of the day looked askance at FORTRAN, disagreeing with the above bullets for various "technical" reasons. One of these was efficiency of the code produced, until it was recognized that it was a rare programmer who could do as well as the designers of automatic code generators. In spite of this resistance, FORTRAN became one of the best examples of improved run-time speed and also the following:

When understandability takes a leap, so does ease of change, and thus productivity.

Anyone racing to build computer programs to solve mathematical problems quickly got on board the FORTRAN train. If they didn't, they couldn't compete and were left behind.

For people building data processing systems, FORTRAN left a lot to be desired. It was cumbersome to work with files, particularly those with complicated record structures. The FORTRAN FORMAT statement is a quick way to get listings of columns of numbers and some alphanumeric data, but there is no friendly mechanism for creating the complex data structures necessary for dealing with large data files. Even the data structure capabilities existing in advanced versions of FORTRAN today leave much to be desired.

Another major problem with FORTRAN is the fall through approach to coding that is a carry over from assembly language coding. Every line depends upon where it falls in the sequence. Labels exist for looping and GOTOs but, in general, one cannot isolate blocks of code inside a subroutine and move them around without great difficulty. An example of a very efficient sorting algorithm, published in the ACM Journal in 1969, [135], is shown in Figure 4-3. This algorithm is very efficient at sorting, using a clever algorithm with a sophisticated mathematical background. But unless one is familiar with the implicit statistical methods for sorting referenced in the paper, one is hard pressed to understand the underlying algorithm.

The example in Figure 4-3 is not meant to reflect poorly on the excellent work of the author. Rather it is a reflection on style and practices of some programmers in that era. Note that, to save time, GOTO's are used to replace DO loops. This accentuates the fall through approach. As an exercise, try putting this example into a flow chart. Note also the Spartan use of identifiers.

This program also exemplifies "economy of expression." A minimum number of keystrokes is required to retype it from the journal - an important consideration of the general programmer. One can also imagine being assigned to make changes to a five to ten page subroutine of this nature - clearly a humbling experience for a rookie. We strongly suggest that economy of expression is *inversely* correlated with the overall life cycle economics of a large software product. We believe that this is easily verified by experimental evidence, e.g., that reported by Fitsimmons and Love, [57], Ledgard et al, [88], and Sitner, [136].

```

      SUBROUTINE SORT(A,II,JJ)

C   SORTS ARRAY A INTO INCREASING ORDER, FROM A(II) TO A(JJ)
C   ARRAYS IU(K) AND IL(K) PERMIT SORTING UP TO 2**(K+1)-1 ELEMENTS
      DIMENSION A(1), IU(16), IL(16)
      INTEGER A, T, TT
      M=1
      I=II
      J=JJ
5   IF(I .GE. J) GO TO 70
10  K=I
      IJ=(J+I)/2
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 20
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
20  L=J
      IF(A(J) .GE. T) GO TO 40
      A(IJ)=A(J)
      A(J)=T
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 40
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
      GO TO 40
30  A(L)=A(K)
      A(K)=TT
40  L=L-1
      IF(A(L) .GT. T) GO TO 40
      IT=A(L)
50  K=K+1
      IF(A(K) .LT. T) GO TO 50
      IF(K .LE. L) GO TO 30
      IF(L-I .LE. J-K) GO TO 60
      IL(M)=I
      IU(M)=L
      I=K
      M=M+1
      GO TO 80
60  IL(M)=K
      IU(M)=J
      J=L
      M=M+1
      GO TO 80
70  M=M-1
      IF(M .EQ. 0) RETURN
      I=IL(M)
      J=IU(M)
80  IF(J-I .GE. 11) GO TO 10
      IF(I .EQ. II) GO TO 5
      I=I-1
90  I=I+1
      IF(I .EQ. J) GO TO 70
      T=A(I+1)
      IF(A(I) .LE. T) GO TO 90
      K=I
100 A(K+1)=A(K)
      K=K-1
      IF(T .LT. A(K)) GO TO 100
      A(K+1)=T
      GO TO 90
      END

```

Figure 4-3. Example FORTRAN program published in CACM in the late 60's.

COBOL - A Big Jump In Data Processing Productivity

Although FORTRAN has come a long way since it was first offered, many problems still exist, causing it to be used less and less each year. The problems described above caused the desire for a new approach early on, particularly for large data processing programs, and a new language was produced in the early 60's to fit the bill. This was COBOL.

It is instructive to leave the scientific community and look at the commercial software industry at a slightly later period in time. After observing the huge productivity gains provided by FORTRAN for the scientific community, computer hardware vendors realized that it was software that was costing clients lots of money. To be competitive, they had to provide support software that improved programmer productivity to compete in the commercial markets. IBM and UNIVAC were leaders in this area, with various languages to cut programmer time. This led to a combined effort by a number of computer manufacturers to come up with a standardized approach to complex data systems that would gain significant improvements in productivity.

The result was the most significant breakthrough in programming for the commercial (business) market - the COMmon Business Oriented Language (COBOL) - circa 1960. COBOL provided major breakthroughs in language design. The importance of using hierarchies to control large complex databases became evident with COBOL's hierarchical data structures and hierarchical rule structures. In fact, code reads almost like English. COBOL hit the New York metropolitan area like a bomb. Software groups that were pressed with huge workloads were looking to get applications up faster, and to enhance them much more easily. This was particularly true when debugging programs written by another author. This split the business programmers into two groups. This time it was the managers versus programmers who were saying that programs had to be written in assembler to be efficient.

But back then, managers of software groups had come up through the ranks and recognized the problem. It was job security. It did not take long for them to hire high school graduates who wrote COBOL code that was better than assembly code produced by experienced programmers, in terms of quality, enhancability, and running times. Programmers who refused to write in COBOL were soon gone, replaced by much less expensive high schoolers. Programmer productivity soared. *COBOL took over 80% of the world's code by the 1980s.*

Learning COBOL was relatively easy. Programmers were soon divided into different skill sets. Some rose to be systems programmers. Others rose to be system designers. Some moved into management. The need to move up the chain was imperative if one wanted to justify a higher salary. Programmers became relatively inexpensive, up until the 1980s.

The COBOL language was developed by experienced programmers to achieve a common goal - build a language that humans could easily understand, one that could read close to plain English, see [15]. To the extent that COBOL quickly became owner of approximately 80% of the world's code for about two decades, it was the most successful programming language ever devised. An October '95 article in Inform, [76], cites studies by IDC, Gartner, and Dataquest that showed COBOL still accounted for over 53% of all applications in the world, 80% of all business applications, 50% of all new business applications, and 5 billion lines of new code added each year. This is because of its ability to improve real economic measures of programmer productivity, where it counts - in the maintenance phase of a life cycle. And these improvements are clearly due to its *understandability* and the ability of its users to map out and access memory in a way that substantially improves run-time speed as well as productivity.

Yet, no language has been more maligned by a vocal segment of the programming world than COBOL, most of whom do not relate to the productivity motive. This is not a new phenomenon. As stated above, in the 1960s when the financial industry in New York City was going through conversions to new IBM-360s, costs to upgrade software were going through the roof. This was because experienced programmers insisted that accounting applications could only be written efficiently in assembly language. What they were really concerned about were armies of high school graduates that were marching into Manhattan and dramatically lowering the cost of building new software using COBOL that was *machine independent*.

Data processing managers had to fight to dislodge their company's software assets from the hands of the assembly language programmers and turn them over to a younger, less skilled workforce who could write code that everyone could understand. In that highly competitive economic environment, it was only a matter of time. The cost of software development and support plummeted, and the leftover money was spent developing more sophisticated applications.

As scientists, we cannot ignore the success of COBOL. We must understand the facts behind its ability to cut costs and improve productivity. Certainly, one cannot contest the readability of COBOL relative to any other language. Greater readability leads directly to understandability. Next, COBOL implemented the one-in one-out control structure advocated years later by Mills, [102]. The objective of this control structure is to eliminate "waterfall" or "fall through" code, providing a hierarchy of blocks of instructions within a subroutine. This additional layer of hierarchical structure serves to increase the understandability of subroutines, a feature that does not exist in other languages. However, as we will describe below, the COBOL implementation hindered certain desired improvements in logical clarity.

COBOL's ability to process data has been unsurpassed. The most important factor in data handling is the hierarchical data structure. COBOL allows a user to organize data structures the way one wants to see it, hierarchically - by logical organization - not by type. Furthermore, What-You-See-Is-What-You-Get (WYSIWYG) in memory. There is no such thing as "word boundary alignment" behind the scenes. Most programmers did not understand the importance of this feature unless they had done sufficient character string manipulation or data processing using a character oriented language. If one has never had this feature, one cannot appreciate it. It's what allows one to do group or subgroup moves from one data structure into another, e.g., moving part of a message or record, defined as all character data, into a template defining each field. It provides for redefinition of data areas so that they can be looked at using different templates or filters without moving the data.

These language features all equate to major improvements in speed as well as productivity. Any language that permits word boundary alignment to go on behind the scenes destroys these features. Until the VisiSoft system described below, no language has provided these data structure facilities nearly as well as COBOL.

Caveats - Some Of COBOL's Short Comings

We must also learn from the weak points of COBOL. The most obvious is the lack of scientific data types and the handling of equations. Although these short comings were overcome with some versions of the COBOL compiler, they were the main reasons for it being shunned by the academic world.

Another problem is the approach to breaking large programs into subprograms. This is a result of COBOL's heritage of sequential batch oriented jobs. Although COBOL provides a subprogram capability, it is not easily used. This has led to extremely large COBOL routines that become difficult to change and maintain.

The reason that COBOL programs are not easily broken into subprograms is subtle. COBOL's sharing of data structures between subprograms by pointer is clearly superior for speed compared to passing individual data elements. However, the mechanism for accessing data structures poses a problem since each structure must be declared in "Working Storage" before it can be used by another subprogram, where it then must be declared in a "Linkage Section." These two classes of declarations impose a constraint that makes it difficult to structure, and especially to restructure, an architecture. It is amplified by the requirement that, in a calling chain, if any routine down the chain wants access to the structure, it must be declared in all routines along the way. One cannot switch or discard the "MAIN" routine without a big upheaval.

As indicated above, COBOL contains a one-in one-out control structure as advocated by Mills. However, the implementation via the *PERFORM paragraph* statement does not preclude the waterfall from one COBOL paragraph to the next, hindering the ability to achieve the desired level of logical clarity. Another implementation "feature" allows programmers to *PERFORM* sequences of paragraphs, further maligning potential clarity. These sequences become especially difficult to follow when they are exited by *GOTO* statements that can jump control anywhere, including the middle of another sequence somewhere else in a large subprogram. This problem is exacerbated by COBOL's unusually large subprograms.

In parallel with the development of compilers was the Operating System (OS) to aid in managing machine resources to run *independent* tasks. We have italicized the word *independent* because it is a key property supporting ease of management - and particularly the allocation - of resources. These facilities allowed computers and software to grow to be huge.

Flow Charts

Because of the difficulty in understanding binary and assembler code, and particularly the instructions for transferring control, programmers created the *flow chart*. When using boxes, diamonds and other symbols to illustrate functions and decisions, a symbol on the flow chart typically encompassed multiple instructions. So the number of lines of code was larger than the number of flow chart symbols. These advantages disappeared with understandable languages and improved control constructs. With COBOL, flow charts became a burden instead of an aid.

The Tower Of Babel - Programming Languages

Although we have only discussed FORTRAN and COBOL, many other early languages had their impact on the software development process. Some of these languages have had substantial followings during certain time periods, but none have matched the long-term success of FORTRAN and COBOL. ALGOL was developed in the early 1960s, partly as an algorithm specification language, one that could be used to specify the details of computer architectures. It was the language used for papers in the Association of Computing Machinery (ACM) Journal. It was the principal language of the Burroughs 5500, one of the earliest time-sharing machines.

SIMULA was another early language, used for simulation. Although hardly used in the U.S., it is referenced frequently because of its discrete event language relationship to the design of operating systems. PL/1 was IBM's answer to provide one language to take the place of COBOL and FORTRAN, a noble goal. However, it was never close to COBOL from a readability standpoint, it had no instruction hierarchy. Also, it had so many options for declaring data that potential ambiguities made programs very difficult to understand and debug.

APL is a good language for solving vector/matrix equations, but is scientifically oriented. PASCAL and BASIC were utilized in the academic community, but never reached the level of use of COBOL or FORTRAN. We will simply mention that each of the U.S. Department of Defense services invented its own language: TACPOL (the Army), CMS2 (the Navy), and JOVIAL (the Air Force). Each language was "justified" based upon the unique requirements of its particular military environment. That is, until Ada came along and the U.S. Department of Defense mandated the use of Ada to replace them all. But, it did not even get as far as PL/1.

BEYOND 1980 - A MORE RECENT HISTORY

As stated above, software technology increased dramatically from 1960 to about 1980. These increases were clearly due to improvements in the programming environment caused by demands of people solving their own application problems. These people were not schooled in programming nor paid by the hour to write code. They forced the sequence of events described above, starting with binary coding and ending with FORTRAN and COBOL, causing huge breakthroughs in language design to improve runtime speed as well as productivity.

Fast forwarding to 2013, the software world has totally changed. One now reads articles based upon measured data describing huge declines in software productivity since the 1980s, see [2] - [4], [6], [15], [19] - [21], [24] - [26], [48], [50], [52], [66], [69], [89], [111], [117], [136] - [139], [141], [146], and [155]. These articles also describe project failures after substantial investments, causing management to hold off on large projects because of the high risk of failure. Projects have been reduced in size, yet productivity still declines.

Yet the power of computers has grown substantially since the early 1980s, and computer costs have dropped dramatically. The advent of the PC has taken the field from large numbers of people sharing a computer, to just a few people sharing a PC with substantial speed and memory to the point where, today, many programmers now have more than one powerful PC available for their own use. Computer clock rates also soared from the 1980s to 2005. Yet, according to Niklaus Wirth's law during that period, "Software gets slower faster than hardware gets faster".

Fast disk storage has grown from 10s of megabytes to terabytes on a PC (100,000 fold). Today it is normal for a programmer's PC to have 4 Gigabytes of cache memory and a terabyte of fast semiconductor RAM. This should have driven most of the programming concerns since the early 1980s to no concern at all.

Another area of major improvement since the early 1980s is the availability of programming aids, e.g., full screen editors with extensive search and replace facilities, special debugging tools, graphical facilities for accessing files, etc. Considering all of these facilities, coupled with dramatic improvements in computer speeds, memory sizes, availability, and project sizes being held down, one would expect programmer productivity to have soared.

In the last two decades, almost all industries in the U.S. have had positive productivity growth, with computer chips being the highest. But in fact, productivity in software has declined more than in any other industry. These facts are substantiated in multiple research studies, see the following references again: [2] - [4], [6], [15], [19] - [21], [24] - [26], [48], [50], [52], [66], [69], [89], [111], [117], [136] - [139], [141], [146] and [155].

Of major concern today is the movement of software development and support offshore to places like India, China, etc., where labor rates are extremely low compared to the U.S. This is clearly the result of the negative changes in software productivity.

Declines In Speed As Well As Productivity

So what has happened to software, a field that was looking great going into the 1980s? Why has it turned around? And why have speed and productivity measures been going down ever since? There are a number of factors, and they all correlate directly to what has become a dramatic decline in the ability to develop and maintain software.

When one compares the history of events since the 1970s, the downturn came about with the upturn in use of the C language (by C we imply the inclusion of C++, C#, Java, Python, and a host of other derivatives of the C language). When one looks at the data on productivity, it is clear that the decline in software productivity correlates directly with the increased use of C-based languages.

If one reads the literature of the 1970s, one finds many papers on language principles affecting programming productivity. These papers made scientific comparisons at the construct level, with arguments why a given approach was better. Coming out of Bell Laboratories, one would believe that the C language was aimed at improving programming technology. Fortunately, most of the true history of C is documented in various AT&T and Bell System Journals. Upon reading that history, one finds that C was not initiated as a Bell Labs project and never intended to be a real programming language. Important principles for improving software described in the literature of the 1960s and 1970s were ignored. See Anselmo, [3].

This has led to tailoring programs for multiple processors (cores), and corresponding facilities for “threads” that are *not* oriented toward large scale (hundreds of) parallel processors. As a result, programmers are back to the EDP board mindset, worrying about timing and synchronization of threads. They are not concerned with planning a huge task to run on hundreds (or possibly thousands) of processors. With the EDP mind-set as a reference frame, automation has not been a concern - at least until now, as complexity multiplies, see [144] and [145]. It’s time to learn the true history of C-based languages.

A New Era - The True History of C-Based Languages

In the first sentence of the preface of their book, **The C PROGRAMMING LANGUAGE**, [81], Kernighan and Ritchie state that "C is a general-purpose programming language which features economy of expression, ... C is not a 'very high level' language, nor a 'big' one, ..." In the second paragraph of CHAPTER 0: INTRODUCTION, it states that "C is a relatively 'low level' language."

When reading the history written by the people from Bell Labs, see [3], [4], [81], [119], and [120], one finds that C was designed to quickly port a game played by a small group of researchers waiting to be assigned to a project. The design goals were made clear by the original designer, Ken Thompson: (1) Use a Spartan syntax to keep the compiler simple to write; and (2) Keep the compiler small to fit in the PDP 7's tiny memory. It was never intended to be a real programming language.

The following quotes are taken from Peter van der Linden's book: *Deep C Secrets*, [147].

C is quirky, flawed, and an enormous success.

- Dennis Ritchie, one of the original authors of C

C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg.

- Bjarne Stroustrup, the father of C++

Apart from the OO concept, there are many reasons why C-based languages are difficult to understand.[†] We refer again to van der Linden's book, [147], on the use of C and C++ in a production environment at SUN, where he questions the placement of the burden of translating code. *Should it be on the programmer, or on the language translator?* On page 64, he states:

"C's declaration syntax is trivial for a compiler (or a compiler-writer) to process, but hard for the average programmer. Language designers are only human, and mistakes will be made. For example, the Ada language reference manual gives an ambiguous grammar for Ada in an appendix at the back. Ambiguity is a very undesirable property of a programming language grammar, as it significantly complicates the job of a compiler writer. But the syntax of C declarations is a truly horrible mess that permeates the *use* of the entire language. It's no exaggeration to say that C is significantly and needlessly complicated because of the awkward manner of combining types."

QUOTES FROM PEOPLE WITH REAL SOFTWARE EXPERIENCE

In 2001, Larry Constantine wrote:

"Continued and rapid growth in the power of hardware has not only enabled new applications and capabilities, but has permitted sloppy, unprofessional programming to become the virtual standard of business and industry. Hardware has allowed the software profession to avoid growing up, to remain in an irresponsible adolescence in which unstable products with hundreds of thousands of bugs are shipped and sold en masse." -- Larry Constantine [48]

He was correct. But as described below, the programmers are now dragging down the hardware guys. In a Software Special Report, [25], Business Week posed the question "Can the U.S. Stay Ahead in Software?" This article described the growing number of software engineers and programmers in other countries whose price per hour is less than 1/3 of their equivalent in this country. Thus, the cost of software development in foreign companies could be much less than their U.S. counterparts, even if they were not nearly as efficient at it.

[†] A common joke is: C is a *write-only* language. One programmer writes it and no one else can read it.

In that same article, Lim Joo-Hong, deputy director of research at Singapore's NCB brought out one of the major factors - "Software only needs people. There is little need for a lot of other resources." In that same article, Edward Yourdon, publisher of the monthly newsletter *American Programmer* in New York warned that cheap labor abroad could begin to make low-level programming jobs in the U.S. obsolete. He was quoted as saying "The only thing that has prevented it from becoming a crisis so far is that the software industry is growing so fast that we haven't seen many jobs taken away." The article further warned "Without such entry-level jobs, the U.S. won't be able to employ large numbers of computer science graduates, further discouraging careers in the field."

But look again. While hardware engineers produce great feats, tearing down barrier after barrier, looming problems in software are hiding behind them. Quoting Marcus Ranum, [117],

"...I see that Microsoft, Intel, and AMD have jointly announced a new partnership to help prevent buffer overflows using hardware controls. In other words, the software quality problem has gotten so bad that the hardware guys are trying to solve it, too. Never mind that lots of processor and memory-management units are capable of marking pages as nonexecutable; it just seems backward to me that we're trying to solve what is fundamentally a software problem using hardware. It's not even a generic software problem; it's a runtime environment issue that's specific to a particular programming language."

Ranum's article is just one example. There is a large group of people experienced in both sides of the computer field - hardware and software - saying the same thing. More importantly, the statistics on productivity show that the software industry has been going downhill every year. But that is not the worst of it. To take advantage of a large number of parallel processors requires a new approach to operating systems. And here, it seems that we can't even get it right for a single processor.

Articles about Microsoft's problems with its new Vista operating system bear this out. In the Sept. 2005 Wall Street Journal, Guth, [69], describes Microsoft's delays in its effort to come out with Vista, a new version of the Windows operating system. It quotes Microsoft executives saying that there was no architecture!

This article was followed up by Cusumano in the ACM, [50], where he talks about the gridlock occurring on the Vista project, stating:

"We now know that the chaotic 'spaghetti' architecture of Windows ... was one of the major reasons for this gridlock. Making even small changes in one part of the product led to unpredictable and destabilizing consequences in other parts since most of the components were tied together in complex and unpredictable ways."

It should be apparent that creating spaghetti code in software is the opposite philosophy of designing independent modules in hardware. If one must replace a hardware module in a production environment, that module must be *maximally independent* from the other modules in the system. Then one can pull it out and replace it with a new module. Equally important is the ability to replace the guts of a module with a new design, using new parts, and still be able to plug that new version into existing systems in the field.

But it now appears that we are already into an era where the computer field is constrained by software problems that present barriers to using new hardware technology. To counter this problem, we must do a reversal on our approach to developing and supporting software. And this approach must be based upon a new paradigm that takes full advantage of all that hardware.

Finally, many experienced people on the sidelines have been saying that the movement to C-based languages and OOP over the past three decades has been a great step backwards for the U.S. software industry. The feature article of a 1994 issue of Upside Magazine interviewed five leading technologists to get their view on the future world of technology in the year 2000. The questions covered broad areas of communications and automation. One of the questions was "What advancement will be the biggest disappointment?" Surprisingly, Gordon Bell, architect of DEC's VAX family, and John Warnock, CEO of Adobe Systems had the same answer - Object-Oriented Programming! Warnock said "I think the whole object thing is a red herring."

The programming productivity problem was highlighted by Paul Strassmann, former Assistant Secretary of Defense for C3I, at a 1992 Ada Symposium at George Mason University, see [141]. There he described the necessary transition of the software industry in his speech "From a Craft to an Industry." He presented the results of a study on the resistance to change in the mode of production of software by what he termed the "loner programmers," the people that every computer installation has come to depend on. He said:

"You can easily identify them. ... They are immersed in their craft, but find it difficult to explain or document it. They usually work late into the night, trying to fix a problem caused by low quality and frequently repaired incomprehensible software. ... They place little reliance on assistance from others and most likely disregard orderly documentation and business practices... The computer code they write is unique, elegant, and usually incomprehensible to others - which explains why they are highly valued as indispensable staff."

As software complexity has grown, current approaches to computer programming using C-Based languages (C, C++, C#, Java, etc.) have taken a major toll on run-time speed. For over two decades Moore's Law doubled processor clock rates every 18 months, helping to conceal the underlying software problems that led to (Niklaus) Wirth's Law: "Software gets slower faster than hardware gets faster." Today we are faced with the realization that semi-conductor expert Jim Meindl's prophecy was accurate, [97]: the Moore's curve for speed has flattened. The doubling of CPU clock rates every 18 months is gone. To make up for Wirth's law, manufacturers are building multi-core processor chips, forcing the use of parallel processing.

Separation of Skills - The Requirement of an Industrial Approach

We submit that separation of skills is the biggest differentiator between a craft and an industry. The industrial revolution not only automated many jobs, it took crafts and turned them into industries. This was most apparent in factories, where different job skills were clearly classified. One did not have to be a craftsman to participate in the production of goods. One could look to a career path that moved up the line as one increased design or management skills. But such an environment does not exist in software. And this is keeping software from moving from a craft to an industry.

With everyone using the same tools there are no comparisons. The result is a lack of production-oriented technology in the software field to support the separation of skills. In every other discipline, e.g., engineering, there is a clear separation of skills based on the application of technology. Chemical, mechanical, aeronautical and electrical engineers all learn to apply math and physics to their application. Why can't they all learn to apply software? Because there are no tools to support them. Those that exist are much too esoteric and not productive.

Why is it taking so long for people to recognize the problem? Having made an investment in becoming proficient in a subject, one does not want to think of that investment as time wasted. It is hard to scrap a skill that took years to learn, one that was supposed to provide significant economic benefits. One does not want to hear that there is a better direction, especially if the alternative involves another learning process, see [46]. This creates a significant inertial factor among proficient C-based language programmers. As described in *Microcosm* by George Gilder, [61], human inertia is the major deterrent to innovation. In a CS faculty lecture on the VisiSoft technology, a professor of software remarked that it was "unprofessional." When asked why he considered it unprofessional, he replied "*Anyone can use it!*" All heads turned.

By following beliefs instead of real history and hard science, almost everyone in the software field thinks that C was a conscious development program at Bell Labs. Unfortunately, no one reads the history documented in their own journals. Bob Allen, Chairman of the Board of AT&T, wanted to compete with IBM in the computer field. He made huge investments in UNIX, and C went along for the ride. Sales & Promotion replaced scientific measures at AT&T. This movement was followed by SUN chasing the workstation and server markets. Customers left IBM, only to return years later when their critical information systems became slow and unreliable. Eventually, the real customers required measurable economic solutions. Internally, AT&T lost control of its own critical switch software - written in C. And SUN worked hard to improve threading, but never delivered on its promise of a parallel processor.

A CASE STUDY - COMPARING SOFTWARE TO THE AUTOMOBILE INDUSTRY

Early on a Friday afternoon in the summer of 1961, two young electrical engineers were driving in an Austin Healey from Ann Arbor, MI to Detroit. They had just completed the first week of a two week course in computer design at the University of Michigan and planned to spend an evening in the city. On the way they had a business meeting with a computer drum and disk manufacturer. Since these two potential clients represented a large organization, the host company engineers were looking forward to engaging in detailed technical discussions. Before the meeting started, someone came in and spoke to the host engineer running the meeting. He turned to the prospective clients, apologized, and said that they would have to move their car off the parking lot. It was a company policy that un-American cars were not allowed in the lot. Finding a spot in the street took some time because of the string of foreign cars.

After the meeting, the two visitors asked about places to go in Detroit. They were told to rethink driving in the city with the Healey - it was definitely dangerous. They could have their tires slashed and the leather top cut off - while sitting in the car. Then one person provided directions to a parking lot that hid foreign cars, indicating that, if they could get there, it would be safe. They went, were safe, had a good time, and got home OK. Later it was made clear how lucky they were.

About that same time, new U.S. technology companies were introducing robots for factory automation. But these were banned from the shop floors of U.S. auto manufacturers. Even the top *quality control* experts - Joseph Juran and J. Edwards Deming - could not apply their skills at improving quality while lowering costs on assembly lines. They were banned from taking data required for the analysis. All of this was justified based upon *job security*. To get elected, the leadership of the U.S. Government stood firmly behind this policy.

But when buyers have a real economic choice and are driven by their own hard-earned money, it becomes a fair ball game. Those who deliver the best results are the winners. Fifty years later, Detroit has been devastated by foreign car manufacturers. One would think that lessons would be learned and things would change. But fifty years is a long time. Hardly anyone thinks that far (two generations) ahead. Those who do are likely to have grandchildren, and considered “too old to understand” today’s world.

One would believe this kind of thinking to be unacceptable in a field like computers, considered at the forefront of technology. But one must look closer to understand the full picture. Thanks to the integrated circuit chip, the computer field is split into hardware and software. Hardware engineers use Computer-Aided Design (CAD) tools to produce their chips, and take measurements to compare data that justifies their designs. Software has become more of an art form, particularly from a user interface standpoint. Who are the buyers today? Just look out the window of your car at the driver next to you. They are probably a part of the fast growing social networking market - texting to their friends, kids, and grandkids.

When comparing markets for investment, one must consider the cost of production in a highly competitive global environment. This may range from labor intensive jobs to high technology jobs. In a nation where labor rates are low, products produced by low skilled jobs will be very competitive. When labor rates are high, low skilled jobs are hard to support, and one must look to alternative sources of revenue. Fortunately, the U.S. has been a major developer of high technology.

Nations must look down the road and pursue markets where they can maintain a competitive edge. As an advanced nation, the U.S. should be striving to dominate markets where demand is on the rise and leadership can be maintained for decades. Ideally, these markets would take advantage of a high technology edge that can be maintained for the long haul as well as used to spawn new markets. The auto industry is one of those markets. As Japan’s labor rates exceeded those of the U.S., they increased shop automation - everywhere they could.

One would think the computer and software markets to be ideal. The number of products depending upon sophisticated computers and automation is growing rapidly, and this trend should continue for decades. This is an area in which the U.S. excelled for many years. However, *since the 1980s, productivity in the software field has dropped - faster than any other industry*. The high cost of building and maintaining software, along with many project failures, has put projects on hold. Large companies are now outsourcing their software overseas, to India, China, and similar countries.

The underlying cause of the U.S. software problem is hidden from public knowledge. It is the result of the same job protection mechanisms that occurred on shop floors in the U.S. automobile industry in the 1960s. However, this time it is the supposed “high-tech” people (the programmers) who falsely believe they are protecting their jobs from lower skilled people, when they could dramatically improve their own productivity and value using CAD systems to develop and support software. But, *they refuse to take the measurements and make the comparisons.*

This problem arose in the U.S. software field in the late 1960s when programmers insisted that business software had to be written in assembly language - a difficult and time consuming environment for building any kind of software. But at that time, software managers understood what was going on (job protection). These managers made the decision to switch to a new programming language (COBOL) where high school graduates could produce and upgrade software faster than PhDs in mathematics using assembly language. Using COBOL, the U.S. software field expanded dramatically.

This same problem is back today. However, programmers no longer aspire to management positions, and instead work to maintain their supposed job security. As a result, today’s managers do not have a good understanding of what programmers do. By the same token, tools used to build software have regressed dramatically since the 1980s and are not much better than the old assembler languages. Programmers hide what they build behind a difficult to understand language environment, protecting their jobs by ensuring that only they know what’s in the code. If a complex software system is to be upgraded, one must go back to the original programmers or spend large amounts of time and money trying to understand what they did.

Yet, the software field is an excellent opportunity where high technology can be used to expand jobs in the U.S. CAD technology now exists that can be used to lower the required skill sets and expand the job market, while at the same time dramatically increase productivity and lower costs. Most importantly, this CAD technology provides the ability to build parallel processor systems that are far more complex than those currently at their limit because of the lack of tools to develop these systems.

The software field is a clear case where new technology can be used to expand a very desirable job market, while at the same time help the U.S. to become much more competitive in a rapidly expanding global market.

The academic community can play a major role in making this happen by injecting a scientific approach into their curricula. This can start with laboratory experiments that generate comparative data. Academia can also develop closer ties with industrial developers. This will require the staff to become familiar with large software systems instead of using snippets of code to demonstrate insignificant points that may be invalid from an overall design standpoint. They must start to understand the economics of supporting large software systems, and the relative costs of saving keystrokes and memory versus the dramatic improvements in time to enhance and run systems in a production environment. *It’s time to face the truth and take the data!*

Detroit Is In Bankruptcy - Is Silicon Valley Headed Down The Same Path?

The prior Case Study compared the U.S. automobile industry to the software industry in the U.S.

The likenesses are obvious, but there is also a major difference.

The computer field is driven by the desire for speed as well as cost.

Before the middle of the last decade, computer clock rates were doubling every eighteen months.

When computer clock rates doubled, software did not have to change. It just ran twice as fast, unless more functionality was added that slowed it down.

Since clock rates have leveled off, computer manufacturers are producing multi-core chips (parallel processors).

To make software run faster, programmers are now told to use more processors, as if 8 processors would make an application run 8 times faster.

With today's approaches to building software, this is far from the truth. The current approach to building software cannot take proper advantage of parallel processors.

To sell chips, manufacturers are now trying to solve the software problems in hardware. But the approach is being driven by the same programmers who do not understand the basic problem.

Many such solutions have been around for years. These involve having the Operating System (OS) make decisions during run-time to make use of parallel processors.

Because the OS has no underlying knowledge of the application, this is the blind leading the blind.

These approaches have slowed down the applications when compared fairly to single processor solutions. But sales promotions are causing the measurements to be ignored.

Silicon Valley is now following the same blind path as the programmers, trying to put into hardware knowledge that only exists inside a specific application.

As described in more detail in later chapters, lack of knowledge of the overall problem is causing hardware designers to head in the wrong direction - one that is preserving the current approach to building software - the root cause of the problem.

Watching this unfold is hard to believe. Silicon Valley is following the same course as the automobile industry. Led by the programmers, Silicon Valley is headed in the same direction as Detroit.

5. *BASIC PRINCIPLES APPLIED TO SOFTWARE*

APPLYING EXPERIMENTAL SCIENCE TO SOFTWARE

It should be obvious from the previous chapters that current approaches to building software are far from a technology based upon experimental science. Journal articles do not contain data representing improvements measured in time. Measures of time - be it wall clocks or stop watches - must be taken as the foundation for experiments to fairly compare software approaches on a scientific basis. Without such a scientific approach, a good solution to the parallel processing problem will never be achieved. More importantly, a great solution will never be accepted.

Without having the data needed to design complex hardware, today's chip reliability would never be approached. Reliable hardware must be designed based upon a huge amount of experimentation over time and careful analysis of volumes of historic data. That is what engineering is all about.

There are a few places in the software industry where this approach is applied. Typically, they are not reported upon. Even if the authors tried, the papers would likely be rejected. Based upon marketing studies, it is likely that these papers would not appeal to their readers. This must change if the U.S. is to take the top position in the software field.

To accomplish this, one must compare speeds on a single processor using different software development approaches. This is best done starting with run-time comparisons using different software approaches on the same machine. The authors have done this and described the experiments in Chapters 17, so others can repeat the experiments. It is not unusual to measure speed differences reaching more than one order of magnitude. What is normal is for the distributions to be quite wide, with the 3σ points exceeding an order of magnitude. Just watching experimenters compare their times, and then their approaches, provides the motivation for use of the scientific approach.

These simple tests are easily expanded to comparisons of productivity. This is because one sees the ability to obtain significant speed improvements using simple theoretical concepts. Once one understands these concepts, fast systems can be built easily. Once built, they are obviously better organized and therefore easier to understand than current approaches. With this in mind, one is prepared to accept comparisons of productivity using relatively simple experiments that achieve the end goal of speed.

The next step is to expand these experiments to the use of parallel processors. This is done in Chapter 18. As shown below, one must be able to comprehend simple concepts that can take advantage of the inherent parallelism in an application. With the correct approach, one can run faster on a single processor than on multiple processors running in parallel. This is another motivator for moving to experimental science where the concepts are so important and easy to use. The first one of these is the use of engineering drawings to represent software architectures. We note that engineering drawings are neither flow charts of code nor block diagrams. This is explained below.

LANGUAGE DESIGN TRADEOFFS

There are a number of tradeoffs that must be considered when designing a software development environment. The principle trade-off is pictures versus words, i.e., when to use *words* and when to use *pictures*. The metaphor “A picture is worth a thousand words” may be a stretch, but it stresses the need for *understandability*. This trade-off may be supported by external documentation. Figure 5-1 provides an example for visualizing 3D graphical transformations. Once one sees the picture, restricting the descriptions of such transformations to words obviously reduces understanding.

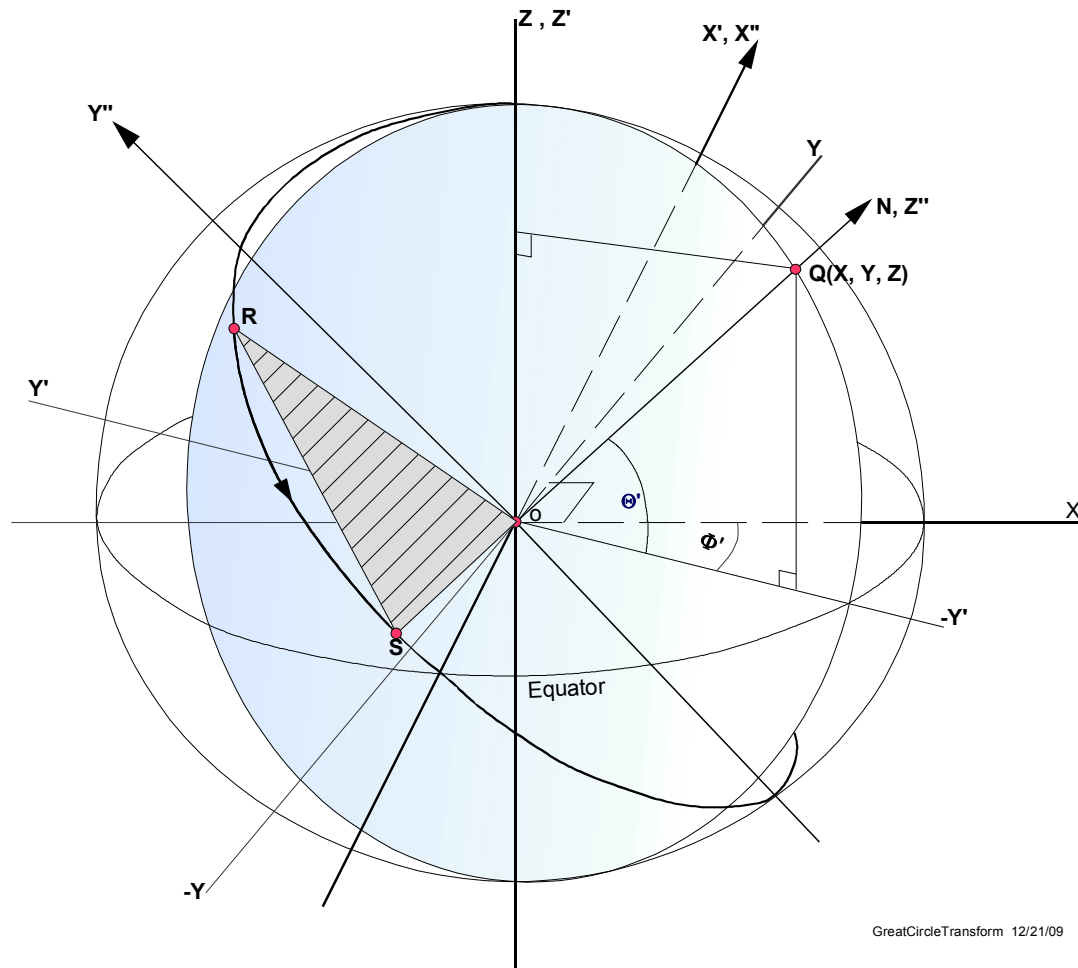


Figure 5-1. Example of visualization of a 3D transformation.

As indicated above, architectural drawings are neither flow charts of code nor block diagrams. They are explicit descriptions of connectivity, i.e., what instructions have access to what data. Using the CAD facility defined here, scope rules are implemented by the architecture, a direct visualization. When using a programming language, everything is defined in the language. Having used this CAD approach, the improved understanding using drawings is as obvious as the example in Figure 5-1. This is described further in a subsequent section.

The next tradeoff is the time it takes to type code versus the time it takes another programmer to understand that code. When building complex software in a competitive environment, minimizing coding keystrokes becomes equally nonproductive. There are numerous articles supporting this statement, see for example, [2] - [4], [6], [15], [19] - [21], [24] - [26], [48], [50], [52], [66], [69], [89], [111], [117], [136] - [139], [141], [146] and [155]. However, considering the current popular software languages and apparent desire to produce Spartan code, this tradeoff warrants further investigation. But to get fair answers requires fair experiments, experiments that are repeatable by independent parties.

In his ACM article in 2000, *The Emperor with No Clothes*, [89], Henry Ledgard quoted W. Edwards Deming who stated “If you can’t measure it, you can’t improve it.” The message carried the same point as that made by David Parnas, [106], 10 years earlier: “Without measures from repeatable experiments, software is not a science.” Although the major initiator of the Computer Science curriculum, Parnas said: “most CS PhDs are not scientists; they neither understand nor apply the methods of experimental science.” Both Ledgard and Parnas are at the top of the list of those knowledgeable in computer languages.

At the very top is Grace Hopper, who wrote the first compiler while working at Univac in 1952. In 1959, after the CODASYL conference started the formal development of COBOL, Hopper’s programming group at Univac spearheaded the language design based upon her own FLOW-MATIC language, see Wikipedia, and Beyer, [15]. Hopper’s belief that programs should be written in a language that was close to English rather than in machine code or languages close to machine code (e.g., assembly language) was captured in the new language, and COBOL would go on to be the most ubiquitous data system language to date. Hopper went on to develop CMS-2, a language for the U.S. Navy that added math and scientific facilities to COBOL. CMS-2 provided the same hierarchical data and hierarchical instruction syntax that contributes huge productivity gains and applies directly to parallel processing. As shown below, software productivity and parallel processing are intertwined.

UNDERSTANDABILITY

A major part of controlling large complex software systems is knowing where functions are performed. This depends on how things are organized. The military is an excellent example of organizing to maintain control in complex dynamic situations. This starts with well-defined hierarchies. If military personnel were only identified by name, organizations would be hard to control - especially for someone new to the organization. As another example, simple databases may be organized alphabetically (no hierarchy). Anyone who has worked with complex databases knows this does not work. Hierarchies are critical.

Engineering drawings of physical systems provide a good example of modular hierarchies. Controlling the design of a huge airliner without hierarchical drawings is impossible. Grace Hopper understood this principle in the design of languages. The ease with which one can create and use hierarchical data structures and hierarchical rule structures is one of the main reasons why COBOL is the most productive data system language.

Hopper also used the combination of hierarchical structures and well designed syntax to greatly simplify the understanding of complex conditional statements. This included setting conditions as well as testing them. Finally, mathematical statements in both FORTRAN and CMS-2 are compatible with mathematical texts. But when language design is driven by requirements to keep the compiler small and easy to write, none of the above logic applies, see [3].

The following quote was stated by Bjarne Stroustrup, the inventor of C++: “English is arguably the largest and most complex language in the world (measured in number of words and idioms), but also one of the most successful,” see [143]. It dominates the world of free trade. Considering the small size of the islands where it originated, its survival is attributed to its reliability. To understand this, consider the military motto, “information is power” - the more information one has to make a decision, the more likely a good outcome. If the information is misunderstood, the outcome is likely to be less than expected. So what are the rules that ensure the reliable transfer of information and understanding?

Language And Information Theory

The two major objectives in communications are reliability and speed. Fast and reliable transfer of information is the goal of information theory, as evolved by Shannon, [130] and others. We start by noting that: *Reliability of information transfers is increased by adding redundancy* (i.e., additional data). This may be as simple as sending the same message twice, or using additional words, such as articles, adjectives, or adverbs. Redundancy is used when writing and reading computer memory. Bits are added to the data being stored to decrease the probability of error when reading it back. In wireless communications, it is not unusual to double the size of the original data stream (redundancy) to ensure reliable transfers. English is considered to have a high degree of redundancy compared to most other languages, implying it is more likely that information is transferred reliably - and key to the survival of its users.

Studies comparing interactive languages have shown that errors increase as statements move from good English to a more terse form, see [88]. Comparisons of COBOL, FORTRAN and C-based languages will typically derive the following programmer reactions: COBOL is verbose; FORTRAN is fair; C-based languages are terse.

The typical misperception is that verbose correlates to slower run-times. In fact, these properties are totally unrelated. Since source language is translated to machine language, the burden is on the machine translator. Reliability and speed can be improved simultaneously. If we are after reliability, verbose is best. If we are after speed, COBOL and CMS-2 are clearly the fastest at handling data. FORTRAN still inverts large matrices faster than C-based languages.

MEMORY

As described in Chapter 1, in the 1960s it was determined that memory costs would never fall below 10 cents a bit. Today, one can buy more than a billion bits for ten cents. The availability of memory has been a prime factor affecting speed since the 1940s. It is still following the Moore's curve - becoming more abundant, cheaper and smaller.

Yet, to most programmers today, increasing clock rates are most important. They cover up the increasing slow software that is being built by programmers with decreasing knowledge of what makes applications run fast. Increasing the clock rate greatly simplifies the software designer's problem.

But the importance of memory has not changed. It is still the single most important computer resource used to gain speed, and exactly what is needed in parallel processing. But programmers using C-based languages have their hands tied trying to map memory hierarchies, and writing code that makes good use of these hierarchies. What's worse is that hardware designers are being led in the wrong direction by programmers that don't understand the problem.

The bottom line is that the memory resource must be optimally mapped and easily used to gain speed. There are times to save memory to gain speed; but most of the time we can use more memory to gain speed. Some cases are offered below, but one must dig into more complex applications to see this obvious fact.

Saving Memory To Gain Speed - Reading & Writing Files

When reading and writing large files, there are special cases where time can be cut by using binary formats instead of readable (ASCII) formats. This is because of the relative size difference of these file types. Anyone with sufficient experience in large data processing applications knows the importance of using binary formats.

To take full advantage of this situation when dealing with very large files (100+ Meg), experienced designers avoid the use of character data, and use minimal sized codes to denote fields with small numbers of states. A simple example is the use of a binary field (1 or 0) to denote whether the light is RED or GREEN. We note that using 5 characters (bytes) to denote GREEN requires 40 bits. Using a 1 or 0 requires only 1 byte (or 1 bit) - a factor of 5 (or 40) difference.

It must be emphasized that the above example is a special case. In large data processing applications, it is not unusual for an experienced designer to make use of various facilities to read, use, and write smaller files 10 times faster than the novice. What's required are language facilities that make these transformations very understandable as well as very fast when the data is brought into main memory. These facilities are described in Chapters 10 through 12.

Using Memory To Gain Speed

The number of cases where one can increase the use of memory to gain speed is large. This was recognized by Mauchley, Eckert, and von Neumann in the design of the Instruction Set Architecture for the first stored program computer. Back then, memory size was hard to expand. This has changed - memory has become abundant. Except for special cases, the current practice of saving memory is the wrong direction. Design of data spaces is critical to speed, especially when using parallel processors. To maximize parallel processor speed, shared memory that changes dynamically can be stacked on multiple processors without waiting, and copied with one fetch when available. Data that does not change need not be shared - it may be copied. Similarly for instructions, especially for library modules that contain only temporary data. With sufficient memory next to the processor, paging can be minimized if not eliminated. These factors provide for nonlinear increases in speed due to the potential shrinking of physical distance between a processor and the memory it must access.

Managing Memory At Run-Time

Applications almost never require all of the memory that may be declared. All of the tables and arrays are virtually never filled. Based upon page faults and tables, the operating system knows when and where memory is needed. Memory need only be assigned as it is used, and this is best done in conjunction with the LRU algorithms used as part of the OS memory management system. The SGI IRIX Operating System was designed to handle any size memory defined in advance in the application software. As memory was required for actual use, if the machine did not have enough hardware to support the requirement, IRIX provided a message stating so. This never happened in practice.

Mismanaging Memory At Run-Time

The use of MALLOCs invokes the question: Who has control? The programmer or the OS? If the machine does not have enough memory, the MALLOC fails, and the OS must pick up the TAB.

The Problem With MALLOCS

Only the OS knows what memory is available at any given time. More importantly, the OS knows when and what type of page-fault occurs that indicates the potential need for more memory to be assigned. Clearly this depends upon the detailed implementation of the memory manager algorithms in any OS. It is well documented that MALLOCs have served to contribute confusion in memory management and is considered a major cause of system crashes.

On the other side of this problem, only the application software designer knows when an area of memory is never going to be used again. Only that designer can tell the OS to *free* that memory area knowing it will never be used again. This requires the ability to name an area of memory, one that may be potentially large, e.g., one used for initialization, and one that potentially holds many different variables and arrays. This also requires the ability to call the OS from the application to FREE that memory resource.

ENGINEERING PRINCIPLES APPLIED TO PARALLEL SOFTWARE

As technology expands, it gets more complex, making it more difficult to create further extensions. This is because the effort required to add improvements can expand exponentially with increasing complexity. Engineers have learned to linearize this phenomenon. Increased complexity may be overcome by pushing the levels down and isolating what is of immediate concern. Simplification of complexity requires applying a number of basic principles that have evolved over centuries in engineering. In addition, end users want increases in run-time speed.

When designing physical systems, engineers must account for problems not encountered in software - for example, parts wear out. Designs must account for what happens when something breaks. What are the possible outcomes? Could lives be lost? Embedded software systems must solve these same problems while meeting difficult time (speed) constraints.

Anticipating problems is a major part of engineering design, starting with the organization of a large number of design and test personnel, and ending with a system that degrades gradually when parts fail. Systems must be back to full operation with a replacement part quickly and easily. This may require redesign of a part to ensure higher reliability. Finding new people who can understand a complex system is much easier when the system design itself is easy to understand. This implies simplifying complexity while maintaining, if not increasing speed.

DESIGN PROPERTIES FOR DEALING WITH INCREASING COMPLEXITY

By its nature, software must deal with ever increasing levels of complexity. Based on the history of engineering technology, significant properties of good design are discussed below. They apply directly to software.

Understandability

As systems become more complex, they become harder to understand. This makes it imperative that systems be designed to be easily understood. *Understandability* of system design is a property that can be measured by tracking productivity in development, test and production. Learning how to take such measures requires many years of experience, working with different types of systems. Without this experience, one must deal with the unknown unknowns. Understandability can be measured in terms of the time and effort required - by people new to a project - to contribute enhancements.

As shown by Ledgard, [88], languages used to specify complex algorithms play a major role in the ability to share understanding. This is inherently a communications problem as described by Shannon, [130]. The language used to transfer information plays a major role in ensuring its correct reception. The English language is known for its redundancy, a major factor in transferring understanding. This is evident in the design of COBOL, the most productive language for information processing. As declared by Grace Hopper, world-wide expert in programming language design (including COBOL and CMS-2), programming languages must be easy to understand - and read like English, the accepted international language, see [15].

Modularity

Another property used to deal with complexity is *modularity*. Systems are best improved or maintained when they are decomposed into separate (independent) modules. This allows modules to be refined or replaced with minimal, if any, changes to the rest of the system.

As proven by approaches to engineering design, modules may be grouped into hierarchies (described below), another critical property contributing to simplification of software design. But the software language must support the ability to clearly delineate the modules.

Independence

The level of modularity that can be achieved depends upon the property of independence. *Independence* implies that modules are isolated, i.e., they are not connected. For modules to be maximally independent, they must be minimally connected. In the case of software, two modules are independent if they share no data. This principle requires a major change in the approach to developing software. Its value is best measured when building software for parallel processors: Modules must be independent to run concurrently on separate processors.

Hierarchical Structures

Going back thousands of years, organizations are best controlled using hierarchies (without hierarchical organizations, the military would be in chaos). *Hierarchical structures* are a critical property of software languages. They support the specification of complex data spaces that are used to simplify complex algorithms. In addition, complex data structures (called Resources) are more easily understood when put into a hierarchy, see Figure 5-2. Finally, hierarchical organization applies directly to the simplification of complex instruction sets (called Processes), see Figure 5-3. The number of levels in a hierarchy must be sufficient to push down the complexity, making the organization of the system easy to understand.

Visualization

Engineering fields (e.g., Architectural, Aeronautical, Electrical, Mechanical, etc.) would be at a huge disadvantage without engineering drawings - precise descriptions of connectivity. Designs also require written specifications. Typically, the crossover point is obvious, as it is in software. Without a language that supports hierarchies and modularity, *visualization* of software architecture, using engineering drawings (not flowcharts), cannot be achieved. See Figure 5-4.

The Separation Principle

The underlying principle supporting modularity and visualization of software is the separation of data from instructions. Known as the *Separation Principle*, [80], separate languages are used to describe data structures (*Resources*) and rule structures (*Processes*). As separate entities, these can be represented graphically - using icons on engineering drawings of software. When a line connects a Process to a Resource, it implies that the Process (instructions) has access to the Resource (data). Interconnected resources and processes can be grouped into elementary modules. Interconnected elementary modules can be grouped into hierarchical modules. One can determine the property of independence simply by inspecting the drawing.

```

USER_FILE_CONTROLS
1  PATH_FILE_NAME          CHAR 66
1  TOTAL_PATHS             INTEGER
1  TOTAL_POINTS            INTEGER
1  POINT_ICON_POINTER      INTEGER
1  PATH_ICON_POINTER       INTEGER
1  RECS_PER_PATH           INTEGER
1  NEXT_PATH               INTEGER
1  END_POINT               INTEGER
1  PATH_PTR                INTEGER
1  PRIOR_POINT             INTEGER
1  MODIFY_PATH_NO         INTEGER

TRANSLATION_CONTROLS
1  PATH                    QUANTITY(500)
2  PATH_NO                 INTEGER
2  PATH_DATA_AREA          INDEX 1
    ALIAS  FREE            VALUE 0,
    ALIAS  USED            VALUE 1
2  PATH_DATA               INDEX 1
    ALIAS  DELETED         VALUE 0,
    ALIAS  EXISTS          VALUE 1
2  PATH_SELECTION_STATE    INDEX 1
    ALIAS  NOT_SELECTED    VALUE 0,
    ALIAS  SELECTED        VALUE 1
2  PATH_MODIFY_STATE       INDEX 1
    ALIAS  NOT_SELECTED    VALUE 0,
    ALIAS  SELECTED        VALUE 1
2  PATH_HIGHLIGHT_STATE    INDEX 1
    ALIAS  NOT_HIGHLIGHTED VALUE 0,
    ALIAS  HIGHLIGHTED     VALUE 1
2  FIRST_POINT             INTEGER
2  NO_OF_POINTS            INTEGER
2  LAST_POINT              INTEGER
2  PATHNAME                CHAR 24
2  PATH_POINT_ARRAY        QUANTITY(1000)
3  PATH_POINT              ICON
3  PATH_SEGMENT            LINE
3  POINT_FINDER            INTEGER

TABLE_FILE_STATE          STATUS  CLOSED
                           END_OF_FILE
                           NOT_END_OF_FILE
                           INITIAL_VALUE  CLOSED

PATH_DATA_BASE
1  PATH_RECORD             QUANTITY(500000)
2  PATH_NUMBER_P           INTEGER
2  PATH_POINT_P            INTEGER
2  WAY_POINT_P             INTEGER
2  BEARING_P               DREAL
2  MOVE_POINT_P            INTEGER
2  LATITUDE_P              DREAL
2  LONGITUDE_P             DREAL
2  ALTITUDE_P              DREAL
2  PATH_POSITION
3  X_POS_P                 DREAL
3  Y_POS_P                 DREAL
3  Z_POS_P                 DREAL
2  DISTANCE_P              DREAL
2  VELOCITY_P              DREAL
2  PATH_ROTATION
3  X_ROT_P                 DREAL
3  Y_ROT_P                 DREAL
3  Z_ROT_P                 DREAL
2  MAX_WAY_P               DREAL
2  MAX_MOVE_P              DREAL

```

Figure 5-2. Resource: PATH_DATABASE.

```

ADD_PATH_POINT
EXECUTE FIND_ADDED_POINT_SEGMENT
EXECUTE PROCESS_ADDED_POINT
EXECUTE DRAW_ICONS_LINES

FIND_ADDED_POINT_SEGMENT
PATH_PTR          = RTG_LINE_INSTANCE_PTR(1)
PATH_PT_PTR_1     = RTG_LINE_INSTANCE_PTR(2)
PATH_PT_PTR_2     = PATH_PT_PTR_1 + 1
IF PATH_PT_PTR_1 EQUALS 1
    SET SEGMENT_POSITION TO FIRST_SEGMENT
ELSE
IF PATH_PT_PTR_1 EQUALS NO_OF_POINTS(PATH_PTR) - 1
    SET SEGMENT_POSITION TO LAST_SEGMENT
ELSE
    SET SEGMENT_POSITION TO MIDDLE_SEGMENT .

PROCESS_ADDED_POINT
IF SEGMENT_POSITION IS FIRST_SEGMENT
    EXECUTE PROCESS_FIRST_SEGMENT
ELSE
IF SEGMENT_POSITION IS LAST_SEGMENT
    EXECUTE PROCESS_LAST_SEGMENT
ELSE
    EXECUTE PROCESS_MIDDLE_SEGMENT .

PROCESS_FIRST_SEGMENT
EXECUTE TOP_SEGMENT_CHECK
EXECUTE WRITE_MIDPOINT
EXECUTE SECOND_HALF
EXECUTE MOVE_SEGMENTS
    INCREMENTING PTR FROM POINT_2 TO END_POINT

PROCESS_MIDDLE_SEGMENT
EXECUTE FIRST_HALF
EXECUTE WRITE_MIDPOINT
EXECUTE SECOND_HALF
EXECUTE MOVE_SEGMENTS
    INCREMENTING PTR FROM POINT_2 TO END_POINT

PROCESS_LAST_SEGMENT
EXECUTE FIRST_HALF
EXECUTE WRITE_MIDPOINT
EXECUTE SECOND_HALF
PTR = END_POINT
EXECUTE MOVE_SEGMENTS

*****

FIRST_HALF
MOVE PATH_RECORD(POINT_1) TO T_PATH_RECORD(1)
CALL NEW_POINT_CALC      *** Expand into Waypoints & Movepoints for
EXECUTE ASSIGN_POINTS

SECOND_HALF
MOVE T_PATH_RECORD(2)     TO T_PATH_RECORD(1)
MOVE PATH_RECORD(POINT_2) TO T_PATH_RECORD(2)
CALL NEW_POINT_CALC      ***B
EXECUTE ASSIGN_POINTS

*****

ASSIGN_POINTS
EXECUTE PROCESS_WAYPOINT
    INCREMENTING WAY_PTR FROM 1 TO NUMBER_OF_WAY_POINTS

PROCESS_WAYPOINT
EXECUTE PROCESS_MOVEPOINT
    INCREMENTING MOVE_PTR FROM 2 TO NO_OF_MOVE_POINTS(WAY_PTR)

PROCESS_MOVEPOINT ***only when at least 2 movepoints
INCREMENT TOTAL_POINTS
MOVE MOVE_POINTS(WAY_PTR, MOVE_PTR) TO PATH_RECORD(TOTAL_POINTS)

```

Figure 5-3. Process: ADD_PATH_POINT (cut off at bottom).

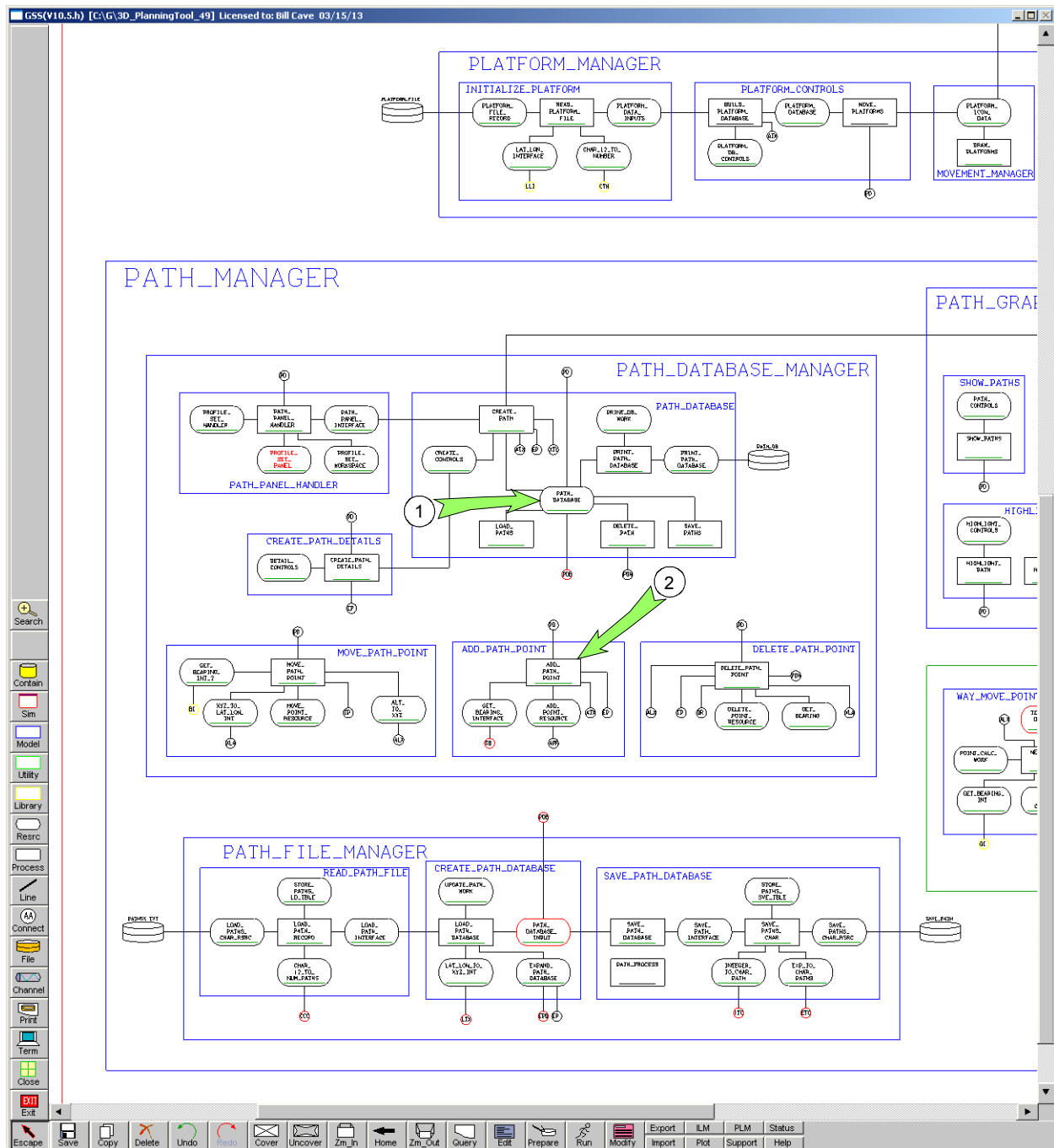


Figure 5-4. Part of an engineering drawing of software showing Resource ① and Process ②.

Spaces & Transformations - The Key To Speed

Likely one of the most successful computer technology innovations over the past 3 decades was the development of the Geometry Engine by Jim Clark - upon which Silicon Graphics, Inc. (SGI) was built. Anyone building complex graphics, e.g., detailed 3D terrain drawn from measured elevation databases, was looking for fast triangle draws. Before SGI, nothing was available that would meet the speed requirements. Scenes took too long to come up on the screen, and changing viewpoints was jerky as well as slow.

Then SGI published its specifications on the geometry engine. Anyone looking for fast draws bought one. No one was disappointed. What was previously done mostly in software was now on fast chips. Order of magnitude speed improvements were typical. The buyers were clearly driven by speed.

Upon lifting the covers on the geometry engine, one was introduced to pipelines of 3D graphical transformations, with various options introduced along the way so as to maximize speed. Speed was obtained by the optimized sequence of complex transformations.

As a by-product, Open-GL became a graphical interface language to the geometry engine. It was designed to make it easy to define the spaces to solve classical 3D geometry problems and maximize the speed of transformations. The speed of triangle draws was in great demand by buyers with a sufficient scientific background to appreciate the technology. Open-GL technology is still at the top of the line.

CHAPTER 6

UNDERSTANDING PARALLEL PROCESSING

INTRODUCTION

The concept of *inherent parallelism* is used frequently in the literature addressing parallel computation. Typically it is in the context of estimating speed multipliers that may be obtained for a particular application when run on a parallel processor. Various authors have described the pitfalls encountered when attempting to derive formulas for such estimates, see [8]. Disagreements often cite improper measures of parallelism, or the serial versus parallel parts of a program.

Most papers addressing speed multipliers cite Amdahl's 1967 presentation, [1], and proceed to support or deny his conclusions. "Amdahl's Law", apparently derived by others from his presentation, defines a speedup multiplier one may obtain from a parallel processor that depends upon the ratio of the serial part to the parallel part of a program. Various authors have argued the validity of Amdahl's law, questioning the definitions of serial and parallel parts in coming up with their own laws, see [68] and [132]. But none of these "laws" takes into account many factors that affect measured speed multipliers, e.g., a limit based upon number of independent modules, regardless of the number of processors, see [43]. Basic issues on single processor versus parallel processor software architectures are generally mistreated. These questions and issues have been further addressed in [44].

This chapter focuses on defining the property of inherent parallelism of a system as it affects the ability to design the hardware as well as software to run on a parallel processor. It also addresses the difficulties of trying to measure such a property. The underlying intent is to shed light on the path to minimizing the time to run a software task using parallel processors.

ANALYSIS OF THE ISSUES

As used in the literature, the term "inherent parallelism" appears to imply the property of a system that allows one to decompose a software task into elements that can run concurrently on separate processors. As used here, a software task is an independent executable in the context of a multi-tasking OS. Upon further reflection, one sees misunderstandings of whether the property of inherent parallelism pertains to a system, or the code that implements it. To understand this, consider that parallel processing has a long history of being used in large scale simulations of physical systems. In that environment, people use the term "codes" to refer to the software underlying the simulations they have built. Because of the complexity of these codes, and the time it has taken to debug them (typically years), changing these codes is considered anathema. When porting these codes to different parallel processors, it is the inherent parallelism of the code that is often referenced, *not the system that the code represents*.

When reading the transcript of Amdahl's talk at the AFIPS SJCC in 1967, [1], and understanding the goal of hardware designers of the time, it is clear that Amdahl was referring to the codes of certain types of problems of interest at the time. His focus was on comparing how different machine architectures of the time (e.g., the vector machines of Cray, the associative memory machines, and the use of separate processors running in parallel) could be used to speed up the solution to these problems. His discussions about the limitations on speedup of a program were based upon the sequential portions of the code.

Our interest in inherent parallelism pertains to the system itself. If we are building software to create a system, or we are modeling a physical system, we want to investigate the inherent properties of that system that can be implemented in such a way as to run concurrently on a parallel processor, and minimize the run-time. We do not deny that laws such as Amdahl's can be used to aid in such measures. However, our goal is to understand how to best design the software to take maximum advantage of the *inherent parallelism of the system*.

To gain a perspective on this problem, we will consider some limiting cases. We start with systems represented by mathematical models, many of which use sets of differential equations resulting in large 2 or 3 dimensional arrays of the type to which Amdahl referred. In such problems one may have to invert large sparse matrices thousands of times. Known solutions to this problem are described in [14] and [71], wherein one generates a symbolic solution in code. This approach minimizes operation counts (e.g., add, multiply, etc.) and eliminates loops. Typically every instruction depends upon the prior one - implying that no instructions can run concurrently. Such code has been designed to run very fast on a single processor and has virtually no inherent parallelism. However, the systems (e.g., electrical networks) represented by this approach may have substantial inherent parallelism. Most of the physical elements operate concurrently with many others. Other approaches to modeling these systems, e.g., using discrete event simulation, generate code that is entirely different, operating much like the physical system when run on a parallel processor. These codes may have a high degree of parallelism.

Next consider the "embarrassingly parallel" case. The usual example is Monte Carlo analysis, performed by running M simulations, each with a different random number seed, concurrently on separate processors. Given that the overhead to start and terminate simulations is insignificant compared to a single simulation, one can expect a speedup factor very close to M . But even in this embarrassingly parallel case, the maximum speed multiplier may be held to M , even when the number of processors, N , may be much larger than M . So why is it embarrassingly parallel? Because decomposition of the problem - into elements that can be run concurrently - is obvious. In fact the code may be a single simulation task that is rerun on many processors in a cluster as separate executables with different random number inputs.

Now consider the case of breaking up the Monte Carlo simulation so that each simulation is run on more than one processor. This decomposition may be difficult, and likely not reflected in the existing code. But if parts of the simulation can be run concurrently, then there is additional inherent parallelism in the system being simulated. This leads to more issues. Can the break up of the simulation be accomplished easily? Once it is done, what is the additional gain in speed of the simulation? Even though the system has additional inherent parallelism, it may not translate into faster running times. To determine if - what may appear to be - inherent parallelism in a system can be translated into additional speed, we must consider factors that affect potential speed multipliers from a parallel processor.

A Summary Of The Perspectives

To review the above perspectives, we considered two limiting cases. The first example was that of systems represented by mathematical models, where each equation depends upon the prior one. This case has no inherent parallelism. The opposite example was Monte Carlo analysis performed by running N simulations concurrently on N processors, each with a different random number seed. Given that the overhead to start and terminate simulations is insignificant compared to a single simulation, and that differences in random number seeds have an insignificant effect on run time, one can expect a speedup factor very close to N. We are concerned with the cases in between.

Speed Laws

As indicated above, the literature on estimating potential speed gains using parallel processors is somewhat divided between those claiming significant improvements and those claiming it is likely not worth the cost. Amdahl's Law defines a Speed Multiplier, SMA, when using a Single Operating System (SOS) parallel processor using the mathematical formula:

$$SMA = \frac{1}{(1-p) + \frac{p}{n}} = \frac{n}{n(1-p) + p}$$

where p is defined as a “percent parallelism” and n is number of processors.

The following are examples. When p is 75%, the Speed Multiplier approaches 4 as n gets large. If p is 95%, and n is 100, the Speed Multiplier is 16.8. Further examples are shown in the plot in Figure 6-1 below. Even when p is 99%, and n is 100, the Speed Multiplier is 50. However, when p is nudged to 100% (most embarrassingly parallel case possible), the curve jumps to a straight line where the speed multiplier is n.

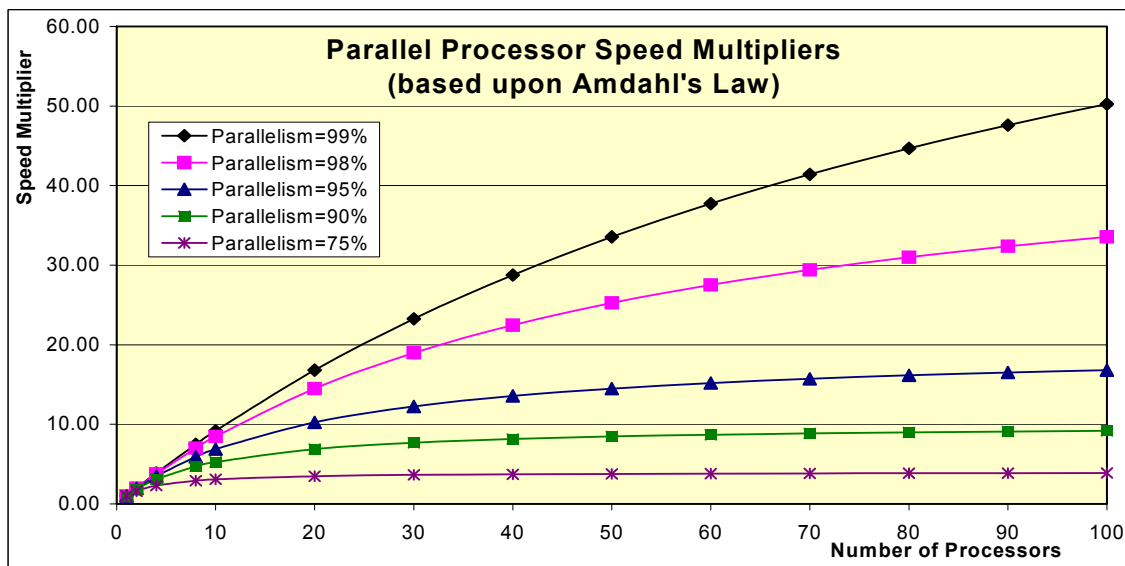


Figure 6-1. Parallel processor speed multipliers based upon Amdahl's Law.

Clearly the curves in Figure 6-1 appear at odds with actual results as well as the basic principles addressed by Gustafson's Law, [68]. But here again, Gustafson's comparison is based upon approaches to breaking up existing code, not the software architecture of the large simulations he used in testing. More importantly none of these papers deals with measuring the *inherent parallelism* in the system or simulation being "coded". We are concerned with useful measures that fairly represent what's "best" in terms of a set of economic choices. This requires a much more detailed analysis as provided below.

Analysis of Critical Factors

We start by considering definitions using the simplified representation illustrated in Figure 6-2. These definitions are independent of both software and hardware architectures. One may argue how components (e.g., those of overhead) should be assigned. This breakout has been selected to simplify the analysis of major factors that can be adjusted to affect speed.

Useful Time (T_U) - Time spent running instructions to perform the functions of the task that would be required on a single processor (including single processor overhead).

Overhead Time (T_{OH}) - Time spent on a parallel processor running excess management or communications instructions that are part of the task, as well as instructions not in the task (OS instructions). OS instructions include those for swapping instructions in the task to be processed by a particular processor, those used to page memory required by task instructions on a particular processor, as well as those for other OS overhead.

Idle Time (T_I) - Time spent running instructions that are not part of the task, with the task itself in a idle state - waiting to be invoked or waiting for communications.

These definitions apply to single or parallel processors except as noted. They assume that no other task is running on the processor (otherwise, results will depend upon what else is running).

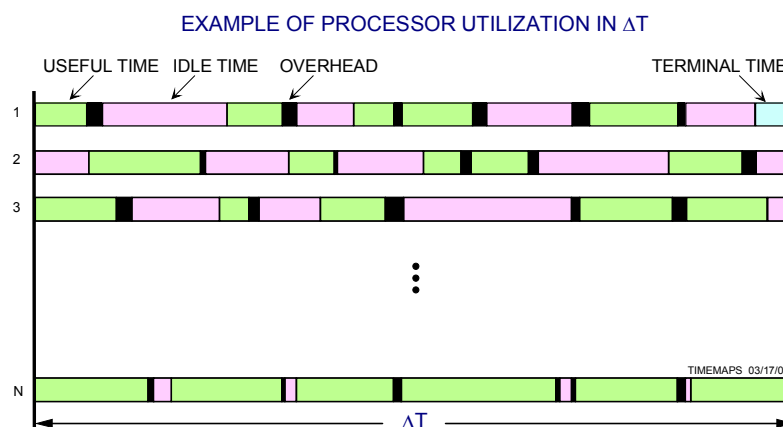


Figure 6-2. Example of processor utilization in ΔT .

Figure 6-2 shows utilization of N processors by a software system running on a parallel processor during a sample period (ΔT). The green area is processor time doing useful work. The black area is processor time spent in overhead functions. The pink area represents time when the task was idle. The blue area is time after which that processor terminates its use (note that processor 1 happens to terminate its participation in this sample period).

We note that the example of processor utilization overlap shown in Figure 6-2 is verified in Chapter 18. This is apparent when looking at parallel processor test results using the Windows processor-utilization oscilloscope.

During the course of ΔT , useful work, overhead, and idle time may be interspersed, as modules on one processor communicate with modules on another. If the green area overlap is insignificant, then little work is done concurrently, and using parallel processors will scarcely shorten the run-time. *This does not imply there is insufficient inherent parallelism in the system. Most often this is due to the approach to decomposition of the software (the software architecture), contributing small speed multipliers when run on a large parallel processor.*

Figure 6-3 shows the relative amounts of processor utilization, overhead, idle time, and termination time grouped together for each processor and ordered by utilization from top to bottom. Clearly the amount of useful work at the bottom of this chart implies significant overlap had to occur, whereas at the top of the chart, there may be scarce if any overlap.

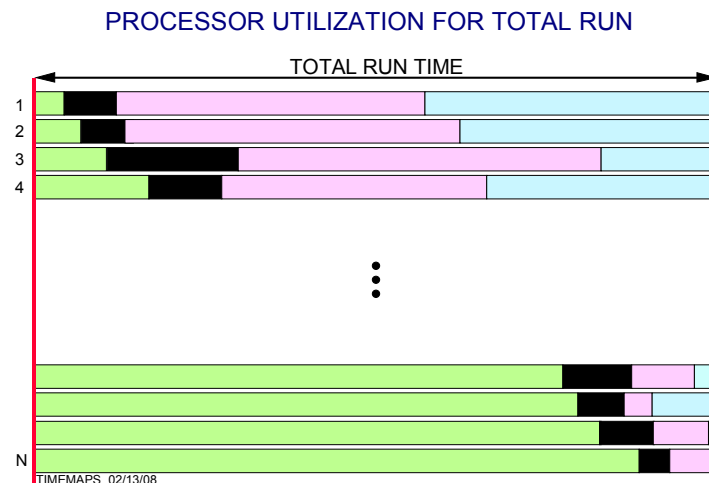


Figure 6-3. Grouped and ordered processor utilization for entire run.

Estimating Processor Utilization Efficiency

If one plots a curve of the ordered *overlapping* useful time spent on each processor (green bars shown in Figure 6-3), one expects different outcomes for different systems, as well as for different architectures of a given system. Figure 6-4 illustrates different possible outcomes using different shades of green. The lightest shade of green has the least useful overlap, the middle shade is in between, and the darkest shade has the most.

If the middle shade follows a straight line, C_m , the area under the curve yields a 50% overlap across processors, implying 50% processor utilization efficiency. This would yield a speed multiplier of approximately $N/2$. In the limiting case, as the curve governing the light green shade, C_l , falls into the origin, the processor utilization efficiency approaches zero. Similarly, as the curve governing the dark green shade, C_h , approaches the upper right hand corner, the processor utilization efficiency approaches 1, and the speed multiplier approaches N . In the limiting case the inherent parallelism in the system has been translated into an architecture such that N processors can be used with approximately 100% useful overlap.

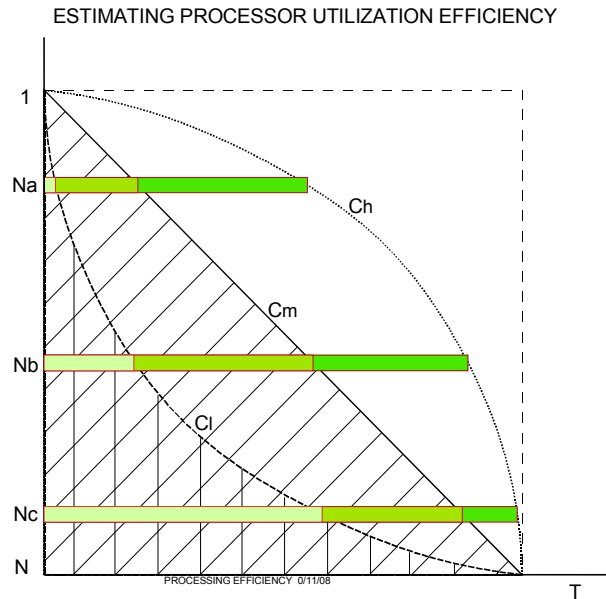


Figure 6-4. Grouped and ordered useful processor time for an entire run.

In general, the areas under the curves in Figure 6-4 represent the corresponding processor utilization efficiency when normalized to the range $[0, 1]$. We note that processor utilization efficiency may be increased by shifting all work from one processor to an under-utilized processor, decreasing the number of processors used. But this will generally increase overall run time if there was useful overlap between the processors. Alternatively, one may shift work from an over utilized processor to an unused processor, decreasing processor utilization efficiency but decreasing run-time if there is useful overlap between these processors. In this case, *one is effectively increasing the resource cost of the run to decrease the time cost.*

Figure 6-5 provides a closer look at processor utilization. Useful time includes time spent performing overhead functions that are also performed on a single processor. Idle time is broken into excess time spent waiting for inter-processor data transfers and control transfers. Overhead time is broken into excess time spent reallocating resources (e.g., for processor assignment, load balancing, etc.) and excess time spent for other parallel processing overhead.

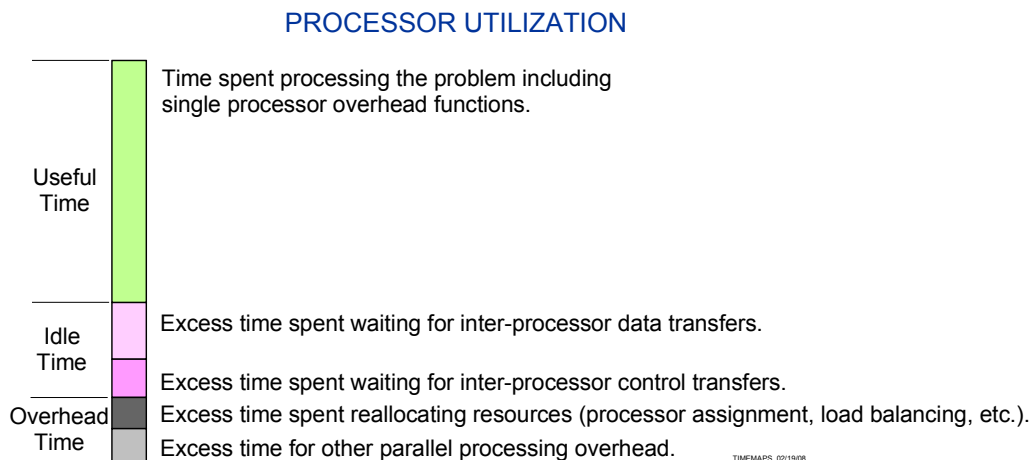


Figure 6-5. Closer look at processor utilization.

Overhead time is affected predominately by hardware and OS architectures, see [44] and [114], and generally minimized with the approach described here. Figure 6-6 provides an even closer look at the idle time. Waiting for both inter-processor data transfers and control transfers can be caused by a lack of inherent parallelism, or by a poor software architecture. These two items are considered the important factors to be dealt with when trying to maximize speed multipliers using parallel processing.

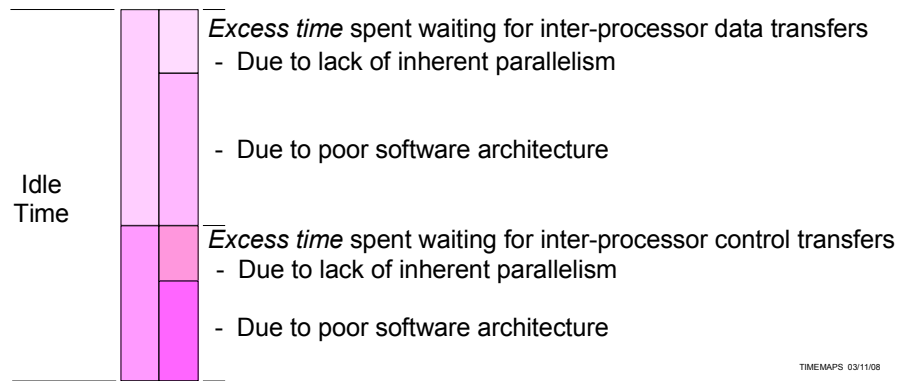


Figure 6-6. Closer look at idle time.

Based upon the above analysis, a processor utilization efficiency of 1 implies no idle time, terminal time, or overhead. This is approached only by embarrassingly parallel systems that are easily broken into totally independent parts. A major concern here is understanding how to minimize idle time for systems that are only partially independent.

MAPPING INHERENT PARALLELISM INTO A SOFTWARE ARCHITECTURE

Whereas other authors are interested in parallelism contained in existing codes, see [86], our interest - as described in the prior sections - is in developing software architectures that take maximum advantage of the inherent parallelism of a system. As shown in the charts above, without useful time overlap, there is no gain in speed. Idle time is likely to be the major cause of reduction in overlap of useful time on parallel processors. As shown in Figure 6-6, idle time is caused by the lack of inherent parallelism in a system or the lack of a software architecture that takes advantage of the inherent parallelism. We are also interested in minimizing the overhead, principally due to the OS. And here there are ample opportunities to reduce the overhead used in current OS approaches to managing parallel processor resources.

We note that the inherent parallelism in a system may also depend upon the input scenario driving it. Consider the example of a transaction processing system, where front end transaction handling is spread over parallel processors, and the back end database handler is also spread over parallel processors. Assume that the database is segmented based on a statistical hit distribution so that the large majority of hits in a sample period are to different segments. When a large number of independent transactions occur in parallel, there may be significant useful time overlap and high processor utilization efficiency. However, if transactions occur sequentially, there is little hope for useful overlap, and parallel processors will provide little improvement over a single processor. When designing such a system, one must look at worst case scenarios that define the design constraints. Such design constraints represent the inherent parallelism of the system.

Discrete event simulation provides significant insights into this problem, see [37]. When designing model architectures to support simulations to be run on parallel processors, one looks for *concurrency in the system* being simulated. By this we imply that *elements of the system must operate independently* of each other. Models built using an architecture that preserves the concurrency properties of the system will also provide for concurrency when run on parallel processors. In both cases, concurrency of operations can be achieved only when those operations are independent. In addition, the scenario driving the simulation will affect the potential overlap in useful processing time, and the corresponding potential to speed up the simulation. When modeling communication systems, message traffic overlap in different parts of the system will vary significantly with the scenario and define the inherent parallelism. Again, one must consider how different scenarios affect the worst case design constraints.

ACHIEVING HIGH SPEED MULTIPLIERS USING PARALLEL PROCESSORS

We now define a parallel processor *Speed Multiplier* as the ratio of processor wall clock time used to complete a simulation or software application on a single processor divided by that on a parallel processor, refer to Figure 6-7. Note that this multiplier will depend upon the scenario used to measure the times, the software implementation of the system or simulation, and implementation of the hardware and OS on the processors. We also note that both software and hardware may be different for the single versus parallel processor. For fair (scientific) comparisons, one must choose a set of well-defined (e.g., worst case/best case) scenarios.

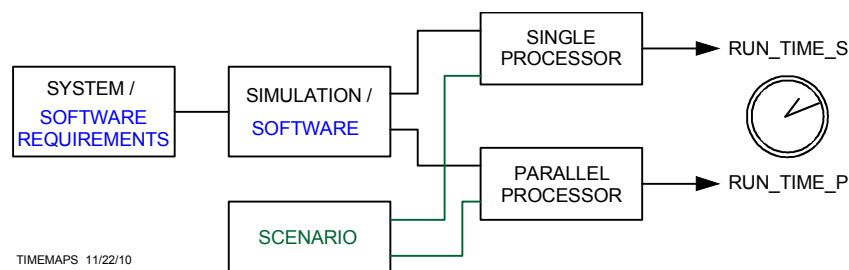


Figure 6-7. Measuring running times on single and parallel processors.

The *Inherent Parallelism* of a system may be defined conceptually as a measure of the percentage of elements in a system that can operate in parallel. Our interest is in building simulations of systems, or software that best implements the specifications of systems for a parallel processor. To achieve high speed multipliers, one must take maximum advantage of the inherent parallelism in a system as it may be implemented in software on a parallel processor. This depends directly upon the software design. Only the designer can translate the knowledge of the inherent parallelism of the system into a software architecture.

Looking at measures of Inherent Parallelism from another perspective, one may consider a theoretical measure as the Maximum Speed Multiplier one may obtain using parallel processors to simulate or implement a system in software. To that end we define *Maximum Speed Multiplier* as the ratio of the smallest single processor time divided by the smallest parallel processor time, where “smallest” implies measures that are dependent on both the software and hardware designs as well as the scenario.

From this perspective, Inherent Parallelism is a conceptual definition that may never be measured directly. However, measures of a Maximum Speed Multiplier may be obtained by observing the smallest running times for software and hardware architectures for both the single and parallel processors for a given scenario.

Comparing Parallel Processor Speed Multipliers

The parallel processor Speed Multiplier (SM) is a measure of the time it takes to run an application on a single processor divided by the time taken on a parallel processor.

$$SM = \frac{T_s}{T_p}$$

This is the most critical measure when determining the value of running an application on a parallel processor. However, it must be analyzed carefully to ensure that it has been determined fairly, see Bailey, [8]. When using this measure to compare different hardware or software approaches, one must use the same frame of reference for comparison, namely the *fastest single processor speed* that has been achieved for that application - independent of the hardware. Otherwise, if we compare software environment A to software environment B, and the single processor time in environment A (T_{SA}), is longer than that in B (T_{SB}), then the software environment that runs slower on a single processor will have a higher SM.

Consider that parallel processor run time T_{pB} is 10 times faster than T_{SB} . Then the speed multiplier SM_B will be 10. If T_{pA} is also 10 times smaller than T_{SA} , then the speed multiplier SM_A will also be 10. Thus, they both have $SM = 10$. But if the single processor running time of B is 10 times faster than the single processor running time of A, B's parallel processor speed will be 10 times faster than A, even though they claim to have the same SM. If A uses B's single processor speed to determine its speed multiplier, it will be 1, not 10. Alternatively, if B uses A's single processor speed to determine its speed multiplier, B's multiplier would be 100.

These numbers may appear to be exaggerated, but VisiSoft has been compared to other software environments by clients and shown to be 10 to 100 times faster on a single processor. Chapter 17 in [4] illustrates this point. The only fair way to compare speed multipliers is by using the same (fastest) single processor speeds. Otherwise, if B uses a single processor for an application that runs faster than A using 10 processors, the economic choice is obvious.

Based upon experimentation (see Chapters 17 & 18), speed multipliers vary widely with software design. When building simulations, one may easily compare different approaches to software design. Using variational analysis, one can derive the important factors that affect parallel processor run-time speed. Generally, the most important factor is modeling a system along physical lines. This is because complex physical systems are designed to be modular. Also, when modules are designed to be independent, they may operate concurrently, representing inherent parallelism in the system. Furthermore, the larger the modules, the more operations that can be performed concurrently and the faster the system. So the obvious design approach is to map these independent modules into models in the simulation. When putting models of these large modules on separate processors, one will likely observe much higher speed multipliers.

The most important conclusions to be drawn from the experiments are the following:

- Unless software module architectures are designed to take maximum advantage of the inherent parallelism in a system, the speed multipliers achieved when running on a parallel processor will not be nearly as high.
- High multipliers cannot be achieved without a run-time environment that receives this architectural information from the development environment and uses it to optimize the allocation of processor resources at run time.

Achieving High Speed Multipliers On A Single Processor

Now consider comparing different simulation speed multipliers using different architectures. The obvious concern when building models to support simulation is the fidelity of the models, i.e., how much detail is in the model. Depending upon the measures of performance being derived from the simulation, the fidelity of the models may have a considerable impact on the results. For this discussion, we will assume that the fidelity of the models used in the different simulations is the same, and that the performance measures are also the same.

If simulation A runs 10 times faster than B on a single processor, it will be selected because of its speed, allowing one to perform fast parametric and sensitivity analyses using Monte Carlo techniques. In fact, this may be the determining factor in whether or not to use parallel processors at all. Although Monte Carlo simulations are embarrassingly parallel, a speed multiplier of 10 may be sufficient to go with the simplicity of a single processor simulation.

More importantly, if simulation B runs 100 times faster on a parallel processor compared to its run time on a single processor, it will only be a factor of 10 faster than A running on a single processor. If simulation A runs 100 times faster on a parallel processor compared to its own run time on a single processor, it will be a factor of 10 faster than B on a parallel processor and 1000 times faster than B's single processor speed. The important information to be derived is that *speed multipliers must consider the differences in single processor as well as parallel processor speeds*. This is best done by comparing the final run times.

This is important since the CAD environment described here is known to produce single processor run-times that are typically 3 to 10 (or more) times faster than the competition for the same application. This is due mainly to the ease of use of large complex data structures, and the speed with which complex data structures may be mapped onto different areas of memory. This is shown by experiment in Chapter 17.

Trying To Map Existing Codes To Parallel Processors

Except for embarrassingly parallel or other special applications, current use of large scale parallel processors typically yields speed multipliers on the order of 10% of the number of processors. This implies that one must use 1000 processors to get a speed multiplier of 100. This is the result of many factors, including the current approach to pulling apart and "parallelizing" code at run-time with little use of knowledge of the system and its inherent parallelism.

Using a good architecture, one can achieve parallel processor speed multipliers that are 60% to 95% of the number of processors. In addition, using large data structures and other techniques, one may achieve single processor speeds that are generally 5 to 10 times faster than other systems. Using this approach, one may expect parallel processor speed multipliers greater than 10 times current implementations.

For those concerned about the cost of rewriting codes, translating FORTRAN to the CAD system described here is reasonably straight forward. Arithmetic constructs are virtually identical. More importantly, it is likely to cost less to rewrite the code than to purchase the number of processors required to achieve the desired run times using current approaches. This would certainly be true if one has determined that an application requires 1000 processors using the current approach, and that number can be reduced to 150 processors using the new approach - a savings of 850 processors. These numbers easily justify a rewrite.

Achieving Order Of Magnitude Speed Multipliers

As indicated above, we are concerned with applications that have a reasonable degree of inherent parallelism (greater than 50% but not embarrassingly parallel). We are also concerned with heavily loaded scenarios where single processor run time is generally the longest.

There are two aspects of speed improvement one can achieve using parallel processors. The first is when one uses more processors to increase the speed of a system or simulation, e.g., using 10 times the number of processors to gain a speed improvement. Depending upon the software environment and design, this may require software changes. In the second case, the speed requirement is fixed but the number of processors is reduced using the techniques described here. In this case, a nonlinear reduction typically occurs since the spatial hardware footprint may be reduced considerably, reducing transmission delays as well as memory boundary crossing delays.

We will use the reduction of number of processors to understand the factors affecting the potential speed improvements. This is a critical point that helps one to understand and take advantage of multiple factors that cumulatively provide large returns, including significant reductions in power consumption and floor space as well as hardware operational costs.

Parallel Processor Utilization Efficiency (PUE)

Comparing running times on parallel processors must include the cost factor, namely the number of processors. This leads naturally to the Processor Utilization Efficiency (PUE), a measure of the Speed Multiplier (SM) achieved on a parallel processor divided by the number of processors used.

$$PUE = \frac{SM}{N_p}$$

Alternatively, the SM is equal to the product of the PUE times the number of processors used. From a software design standpoint, PUE is the most critical factor determining the speed with which an application runs on a parallel processor. Again, PUE must be calculated fairly using the same (fastest) single processor speed to compute the SM for the application.

The PUE depends upon the average amount of useful work done on each processor within a ΔT_{\max} window, [114]. When the amount of work done on each processor is highly varied, many processors will have large idle times, and the resulting PUE will be low. When the useful times spent on each processor are all close in size, the average idle time in a ΔT_{\max} window is typically much smaller rendering the PUE much higher.

Factors Affecting The PUE

The major factors affecting the PUE are the following.

- **Single Processor Speed Multiplier** - This is the difference in speed using the VisiSoft CAD environment versus current “advanced” software development environments on a single processor. This implies taking advantage of the language properties. Based upon the tests in Chapter 17, the typical speed multiplier is from 10 to 100.
- **Mapping Of Inherent Parallelism** - Using VisiSoft, one can design software architectures that produce an optimal map of the inherent parallelism of a system into Independent (IND) modules that take maximum advantage of the hardware architecture (see Chapter 19). These properties affect others, e.g., Processor Utilization Efficiency.
- **Better Use Of Chip Space** - VisiSoft maps IND modules into separate processors and eliminates designer concerns for thread synchronization. The use of Inter-Processor (IP) resources between IND modules eliminates any concerns about coherency as described in [58]. Sharing memory with a server eliminates the need for DMA channel interfaces to external devices. Stack facilities and complex instruction caching are eliminated. All of these serve to simplify the chip design allowing more memory close to the processors, further reducing swapping and paging. This will further improve processor utilization efficiency.
- **Processor Utilization Efficiency** - This is estimated to range from 85% to 95% versus the typical 7% to 10% encountered when using large numbers of parallel processors. VisiSoft IND modules are generally large and remain on a specified processor, minimizing swapping and paging. Changes in scenario loading can be neutralized by migrating modules at run-time to use smaller numbers of processors while meeting speed constraints. Coupled with improved mapping, this yields another factor of 10.
- **VisiSoft Parallel OS (VPOS) Speed** - This is the difference between a standard OS and the VisiSoft design for VPOS. It is estimated that using VPOS with the above factors provides speed multipliers between 100 to 1000 times faster than existing approaches.
- **Distance Factor** - This is the result of the reduced distance between processors and memory due to the above factors, producing the same speed multiplier with a reduced number of processors. Note that this reduction is nonlinear with distance. If the number of processors is cut by a factor of 100, one may discover an additional speed up factor of 10, just due to the reduced distance between processors and memory.

As indicated above, these factors depend upon other factors, e.g., the size and intensity of the scenario. Numbers must be derived from experiments using actual applications. The estimated ranges above run from over 100 to 10,000 times faster, and are considered to be rough but reasonable based upon prior experiments with the type of simulations considered.

When minimizing the number of processors, final speed multipliers are affected by all of the above factors. Our interest is in comparing complex simulations for analysis and design applications using current technology versus that developed using VisiSoft. Consider an estimate of the reduction of number of processors to achieve a given speed as provided in Figure 6-8. These are considered representative for the types of simulations considered. We note that the reductions shown in the figure do not include potential improvements in processor utilization efficiency due to improved chip design, but do assume reasonably loaded scenarios.

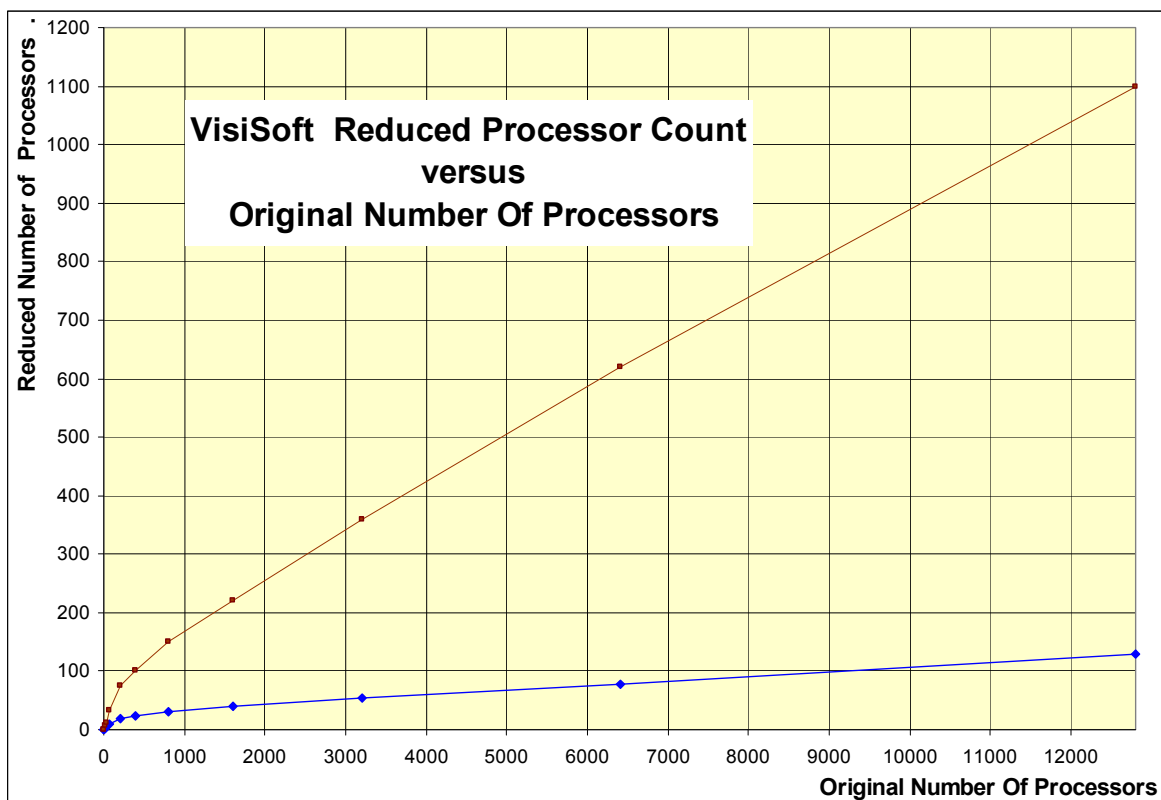


Figure 6-8. An estimated reduction of number of processors to achieve a given speed.

The Distance Factor Versus Speed Of Communications

The speed with which we can communicate today has increased dramatically over the past few years. Gigahertz channels are now available compared to megahertz channels not too long ago. However, understanding what this means with respect to parallel processing requires a careful analysis. Figure 6-9 below illustrates the comparison of a 10 GHz signal to a 20 GHz signal. Doubling the frequency generally implies doubling the number of bits that may be packed into a time pulse. Thus, we may receive a large set of data in a short time period.

This is particularly useful when moving large databases. The move from 1 Gigabit to 10 Gigabits per second implies files may be transferred 10 times faster. File transfers that may have taken 60 seconds can now be done in 6 seconds. This is clearly a huge increase in transfer speed. But what does this have to do with parallel processing?

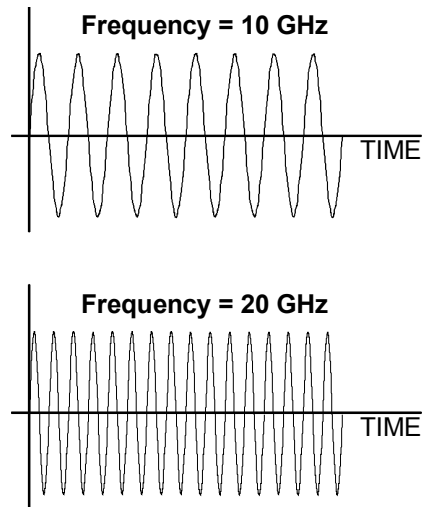


Figure 6-9. Measuring communication speeds between parallel processors.

Data Transfers Inside Parallel Processors

Data transfers inside parallel processors are typically quite different that file transfers over the internet. When comparing file transfer times over the internet, one is usually comparing the time in seconds, or response times in milliseconds. A raw megabyte file contains 8 megabits. However, it is not unusual to double that size when adding coding techniques to ensure reliability of transfers, implying 16 megabits. A 10 megabyte file would require the transfer of 160 megabits, taking 16 seconds over a 10 MHz channel, or 16 milliseconds over a 10 GHz channel, clearly a huge improvement in speed. However, a string of bits taken from computer memory must be encoded and put through a signal processor that transforms bits into waveforms as illustrated in Figure 6-9. More importantly, 20 GHz corresponds to 50 picoseconds in the time domain, but a bit may take 20 times that. In any event, the wave cannot travel faster than the speed of light, approximately 1 foot per nanosecond.

Inside a parallel processor, one is normally concerned with much smaller data transfers, typically on the order of 10 to 100 Kilobytes or less. One is not concerned with how many bits can be put into a packet, but how fast one can transfer the bit string through the computer. When working with a single processor, one typically deals with memory transfer delays on the order of a nanosecond to and from level 1 cache memory. Such delays are typically not noticeable in a communication system. Delays between a processor register and level 2 cache will likely not be that much larger when on the same chip. However, going from chip to chip, one may find an order of magnitude increase between level 3 and level 1 cache. Although the level 3 cache may still be considered very fast, one may lose a factor of 5 to 10 in speed of instruction processing.

Chapter 16 provides a chart of memory boundary crossing delays when moving from the same chip to the same board to the same tray and finally to a different rack. When comparing large High Performance Computers (HPCs), it is not unusual to find 50,000 processors taking up a room full of racks. This is illustrated in Figure 6-10. Distances, D , between processors go from centimeters to many meters. But one cannot simply use the speed of light to estimate the time delays. One must account for the delays through communication switches as well as the memory boundary crossing delays described in Chapter 16.

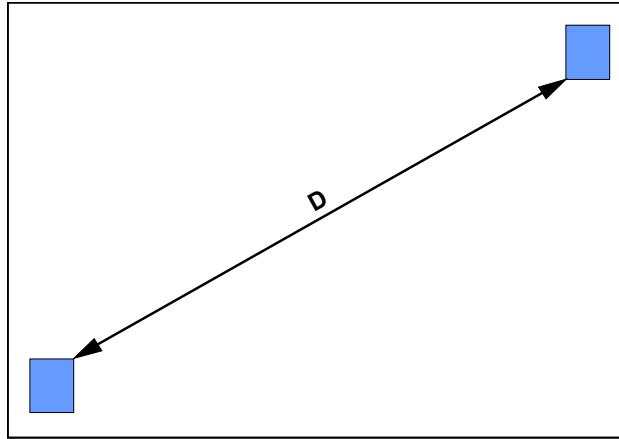


Figure 6-10. Measuring communication speeds in high footprint parallel processors.

Embarrassingly Parallel Applications Versus Those That Are Not

When dealing with embarrassingly parallel applications, one is generally running independent tasks on separate processors so that communication delays between processors are not of concern. The typical application is Monte Carlo Analysis, where each processor runs the same analysis or simulation starting with a different random number seed. Comparison of results is done via a separate data collection and analysis task after the parallel processing run. Such embarrassingly parallel applications may be run on clusters of separate computers, each with its own OS. This implies that communications between computers must pass through the protocol layers of a communication system on each processor, a time consuming process.

When running applications that are not embarrassingly parallel on a large footprint system, it is common for Processor Utilization Efficiencies (PUEs) to fall below 10%. This implies that one must use at least 100 processors to achieve a speed multiplier of 10. If one can improve single processor performance by a factor of 10, then one processor can do the work of 100 parallel processors. This implies building processor chips that are powerful and have access to large amounts of memory (many gigabytes).

TEMPORAL INDEPENDENCE AND DECOMPOSITION

As indicated above, two simulation models may be considered independent during a time period that is sufficiently small to provide a valid representation of the real system. If this time period is large enough, then sufficient processing may be done by running these models in parallel. However, if the effects of a model in one processor on that in another are too tightly coupled in time, one may not benefit from running on separate processors (the results may be valid, but the speed gains insufficient). This problem occurs in the design of systems as well as when modeling such systems. We must answer the question: Given a system to be designed or simulated, how can it be decomposed into separate modules/models so as to maximize speed while maintaining proper operation - or simulation validity - when using a parallel processor? To answer this question, we must address the temporal aspects of these systems.

Synchronous Versus Asynchronous Systems

When designing systems, e.g., digital communications and computers, one is typically concerned with throughput or speed of sequences of operations. Design of modern communication systems is based upon Shannon's Theory of Communications, [130], where one must obtain the correct signals (bit streams) in the presence of noise using a limited bandwidth. The theory applies to the design of electronic circuits for computers as well as digital communication systems. Signals flow through logical networks to produce a resultant sequence. Speed can be improved by processing signals that can be split into parallel parts. Significant experience recorded in the design of such systems has divided the problem into two segments: synchronous and asynchronous.

As advanced by the U.S. military, mobile radio communication networks are by far the most complex to design. They consist of a network of radio nodes where all nodes are moving and connectivity (who talks to whom) is constantly changing. When properly designed, connections can be supported through changing paths. We note that cellular networks have a fixed infrastructure, since all nodes are fixed except for the terminals attached to a base station (problems still exist with handoff from base station to base station). The most advanced forms of radio networks use Time Division Multiple Access (TDMA) techniques, wherein signals are pulsed within time slots. Without going into details, sets of signals are processed asynchronously within a time slot and decoded synchronously at the end of the slot. All units must maintain a sufficiently accurate clocking system. Although some have argued theoretically that systems may be built without synchronization, when tested, they have never been made to work.

The same approach is used in designing digital computers, where signals are processed asynchronously through parallel logical structures (gates) with the results derived in stored elements (flip-flops) on a synchronous basis (using a clock pulse). Again some have argued theoretically that systems may be built without synchronization, but when tested, they are not sufficiently reliable.

An approach to solving this problem for parallel processing was embedded in the Time Warp Operating System (TWOS), see Rieher [121], which has been renamed to SPEEDES. This experimental operating system was reported upon bi-annually in the late 1980s to early 1990s at the International Conferences on Simulation. It allowed threads to get out of synchronization with the simulation clock with the idea that they could be "reprocessed" to resynchronize results. However, it was determined that in partially independent cases, unscrambling the resulting chaotic states was virtually hopeless, and validity of results was clearly lost.

To support the ability to take advantage of potential "slack times," wherein the ordering of events does not affect the outcomes of a simulation, VisiSoft provides a ΔT_{\max} synchronization facility that is automatically invoked by the Run-Time System (RTS). As described below, this facility allows the user to schedule events to occur based upon the simulation clock (or real-time clock) so that events may occur asynchronously within a time interval (ΔT_{\max}) at the end of which all events are held for synchronization. This facility is integrated into the overall scheduling system that supports the use of event threads that may run in parallel, independently, when contained in Independent (IND) modules/models. This approach has been shown to work reliably, even with systems sensitive to such variations, when a cross-over point can be established by testing and comparison to single processor simulation outcomes.

The next section describes the use of ΔT_{\max} to speed up parallel processor simulations while ensuring their validity. To do this, one performs statistical tests that produce distributions which can be compared to live test or single processor simulation results. Such tests are necessary to determine validity of single processor simulations. We must emphasize that the simulated tests must be preformed in the same manner as the live tests, else the statistics will have questionable meaning. Conversely, one cannot design a system using simulations unless the designs are evaluated in a manner that accounts for all of the variations that affect a live system. This is a common cause of live system failure when designs are based upon simulations that do not account for all of the potential variations (they are typically unknown unknowns).

ENSURING VALIDITY OF SIMULATION RESULTS

Test results can be presented in many ways. Field or laboratory testing is subject to the Uncertainty Principle, and the properties of a system being tested are typically presented in terms of a distribution derived from a sufficient number of measurements. We use V to denote a vector of values measuring the properties of a system under test. At the end of a series of tests, V is represented as a set of distributions, one for each property or element, V_i , of the measurement vector. Typically, these test distributions, D_t , can be characterized as illustrated in Figure 6-9.

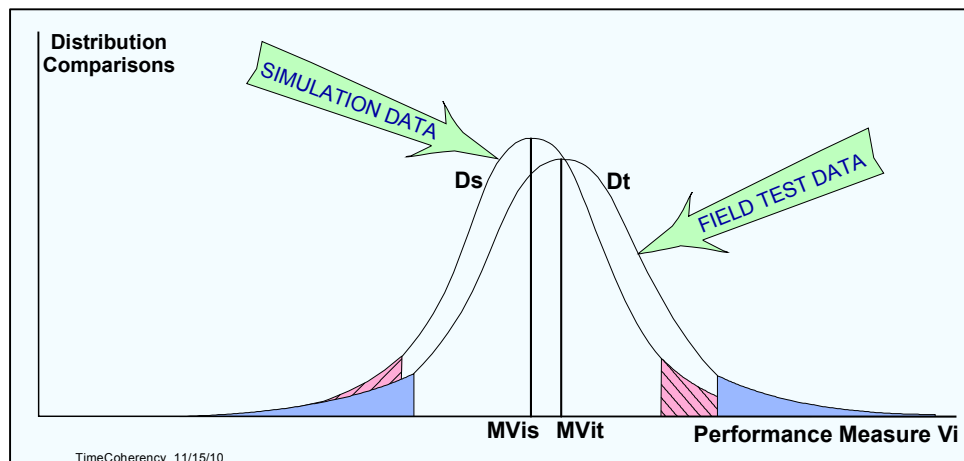


Figure 6-11. Illustration of the distribution of test results for performance measure V_i .

When field or laboratory testing is prohibitive or expensive, one typically resorts to simulation. One must then assess the cost of obtaining valid results from a simulation. When running a single processor simulation, one will get the same results from every run unless it is taking in different data or data generated in real time. Typically this does not coincide with a real world test that is subject to variations in parameters of both the system and the test process. To provide a more accurate view of simulation results, model parameters that may vary in real operations are drawn randomly from predetermined distributions representing known or anticipated variations. These are used to generate the distributions, D_s , also illustrated in Figure 6-11, and are used to analyze validity of simulation results, see [44] and [114].

To analyze the effects of variations, one typically runs Monte Carlo simulations, whereby the simulation is run a sufficient number of times, each with a different random number seed, to produce the distribution, D_s , of results. Then one can compare the distribution of results from the simulation to that of a valid test set. If the distributions are deemed to be sufficiently close by the “validation committee,” then the simulation can be used as a valid substitute for field or laboratory tests.

Validating simulations may be difficult, but typically not more so than validating data from complex field or lab tests. Simulation validity can be achieved on a model-by-model basis and by comparing the results of simulations to those from a reduced set of laboratory or field tests. In many cases, a subset of models may have been previously validated and can be reused in different simulations. Regardless, validating a single processor simulation is outside the scope of this treatment, so we will assume that a single processor simulation exists that produces valid distributions for the measurement vector. We are concerned with obtaining sufficiently valid results when moving from a single processor to a parallel processor environment.

Validating Parallel Processor Simulations

Based on the above, we will start by examining approaches to determining validity. This implies investigating the requirements for completeness and consistency of the measures of those state variables that determine the validity of a simulation when compared to accepted test data. If the measured results are not complete, i.e., some measures are lost or incorrect, the results may be considered invalid. If results are not consistent - as a function of time - with the test data or prior satisfactory results, then they are likely to be invalid. All of these determinations require judgment because of the statistical variations that occur in actual systems. This task can be substantially simplified when one has a validated single processor simulation, and the ability to investigate the potential loss of validity when running that same simulation on a parallel processor.

We must next gain an understanding of the causes of the loss of validity of results. When using standard software approaches, one may allow two or more processes residing on two different processors to share memory. Unless they have a synchronization protocol between them (sometimes referred to as a lock) they may both be running concurrently and accessing that memory. If one of them writes into that memory, changing a particular field, and the other reads that field assuming it has not been changed since it was last accessed, then the consistency of results that depend upon that data is lost. Furthermore, any changes thereafter that use that data will produce potentially inconsistent results.

This is a common problem in communications systems, with known approaches to solve it. The best solution does not slow the transfer of information nor burden the designer - except to ensure that the system requirements are met. Given that we can solve it for simulation, the software solution follows directly.

When modeling a communication system, multiple sources may send messages at the same time to a receiver. Given that the receiver can only receive one of those messages, then a decision must be made as to which, if any, can be received. In the case of simulation, this is a modeling problem where the receiver must determine if it can receive any of the signals (they may all be lost) or if one is sufficiently strong to be heard over the others. This becomes a synchronization problem, where the receiver must make a determination using information about multiple signals. When modeled properly, the results will match the live test data.

The ability to do this fast is built into VisiSoft Inter-Processor (IP) resources. These must be used between Independent (IND) modules. These limit write privileges to a single process and provide copies of the resource to all readers. When a process starts to run, its IP resources are updated. When it is running, its IP resources cannot be changed, unless it has write privileges. The problem then becomes a synchronization problem.

Just as in the real communication system, the correct results are obtained by synchronizing at a given point of time when all signals must have been received. At that point, with all of the necessary data on hand, a decision is made as to which one was received. This decision is based upon the specifications of the system as defined by the designer.

Solution to the validity problem is not limited to discrete event simulation. It can be used in software, and also applies to real-time systems. In this case it assumes that the processors are sufficiently fast to make the proper decision within the synchronization period. In the end, the decision depends upon the specifications of the system as to what message was received.

We must emphasize here that we have seen various theories on how best to maximize speed while ensuring validity of results. In our experience, it comes down to measures of speed as well as the basic validity measures. What is important is not the theory, this may be argued forever without a clear outcome. *What is critically important is the comparison of carefully measured results of actual experiments.*

Simulation Time Synchronization And Validity

In discrete event simulation, processes are scheduled at specified (event) times in the future, with the anticipation that - by design - data accessed by these processes will be correct at those scheduled times. On a parallel processor, loss of validity may occur due to loss of time synchronization on different processors, where time is simulation clock time.

If a single processor version of a simulation is producing valid results, then it can run on a parallel processor and produce the same results provided the following is true: Processes running on different processors and sharing data run in the same sequence as they would on a single processor. But this is not a necessary condition for validity of the results. In live experiments or field tests, ensuring such sequences always occur the same way is typically intractable if not impossible. This is why it is common for single processor distributions to be more narrow than the corresponding field tests, see Figure 6-11 above.

Using distributions to state this more carefully, if MDs is a positive number representing a measure of the accepted value of the single processor distribution error, and MDp is a positive number representing a measure of the parallel processor distribution error, then the deviation between them may be defined as follows:

$$DEV = |MDs - MDp| \leq \epsilon .$$

Then, to ensure that an acceptable value (less than ϵ) is obtained for the deviation, one must ensure that the deviation of a parallel processor sequence of events does not fall outside some ΔT . Thus one must find a value of δ such that if ΔT is less than δ , then DEV is less than ϵ . We will refer to this maximum time value to allow changes in sequence as ΔT_{max} . This is effectively a synchronization window that, when decisions are made inside this window, the results will be valid.

To validate the results of a simulation, one must compare the distributions of the elements of the measurement vector from the parallel processor simulation to that of a valid test set, or valid single processor simulation. Validity can be achieved without having the same process sequence. The process sequence is typically varied in the single processor case simply by varying the random number seed. As stated above, maintaining a strict sequence (temporal consistency) is not a necessary condition for validity. What is important is to know what will produce valid results from a simulation, and what happens when we move that simulation from a single processor environment to a parallel processor environment.

In a parallel processor simulation using the General Simulation System (GSS), see [67], one can ensure that the simulation clocks on each processor do not differ by more than ΔT , a parameter specified by the designer. We note that, using GSS, the simulation clock units can vary from picoseconds to days within a simulation. This allows the designer to set ΔT to the maximum value, ΔT_{max} , that still ensures validity of results. This is a key factor in obtaining higher speed multipliers from parallel processing, reducing idle time of processors waiting for clock synchronization. Figure 6-12 shows the effects of increasing ΔT in two different simulations, A and B.

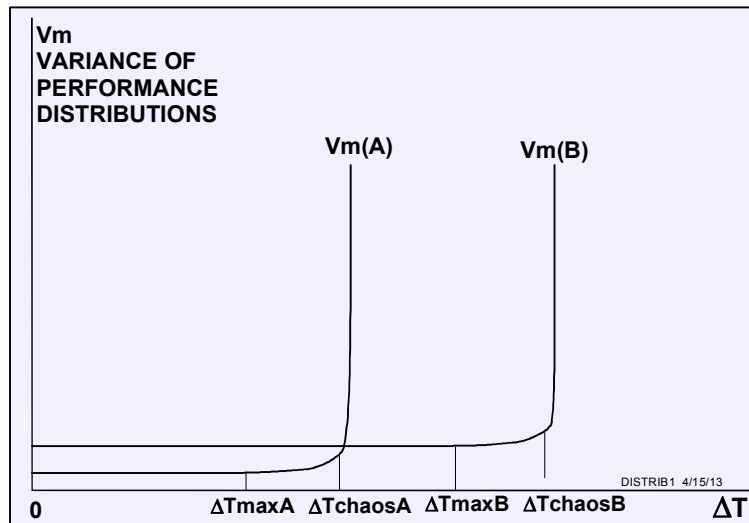


Figure 6-12. Curves showing the effects of ΔT in two different simulations.

As ΔT is increased, a point is reached, ΔT_{chaos} , where the simulation becomes chaotic as shown by the rapid increase in variance of the distributions at that point. The shape of these curves has been validated with actual test data, see [114]. The motivation for increasing ΔT is improvement in parallel processor utilization efficiency and the resulting speed multiplier that is obtained. References [44] and [114] also show how the architecture of a parallel processor affects the speed multipliers.

Modeling Nonlinear Dynamic Systems

To better understand the temporal factors affecting processor utilization, and more importantly idle time, we consider models of electrical networks. Such models are often used as analogs of mechanical or fluid dynamical systems. They are typically characterized by systems of nonlinear differential equations. When attempting to decompose these systems so separate models can be run on separate processors, one must be concerned about the convergence of nonlinear models at a given time step. This is generally determined by the time constants of the system and the nature of the nonlinearities.

In the case of linear systems, one can determine simple models (equivalent electrical circuits) that present an equivalent of the input to the next stage. This allows large models to be decomposed into smaller “decoupled” models. In these cases, ΔT_{max} may be determined by the amount of detail one wants to see in the output response, becoming in effect a sampling rate. The minimum sampling rate, ΔT , is typically calculated from the shortest time constant in the system and Shannon’s theory, [130]. In general, separate models will have different time constants, requiring different sampling rates. One may use the fastest sampling rate based upon the resulting time constants of models that interact. In the end, it must be determined by the validity of the results.

Many systems of interest are highly nonlinear. In these cases, one may need to iterate to gain convergence at each time step, as well as use a high sampling rate. Using discrete event simulation, one may be able to model the physical system in such a way that the nonlinear effects are determined using a variable sampling rate, one that depends upon the current state of a model. In any event, the sampling rate is determined by the requirements on model validity as defined above.

Given that models may be split across processors to run concurrently, they will run until the next scheduled process time exceeds the selected ΔT synchronization point. At that point, all must wait for the slowest one. This causes idle time on the rest, reducing processor efficiency. Processor utilization efficiency may be raised by putting multiple “slow” models on a single processor, using fewer processors. However, if there are a sufficient number of processors, using them less efficiently may be the easiest as well as the fastest approach. This becomes a time-cost trade-off.

SUMMARY

Upon analyzing the issues relative to inherent parallelism in systems, we have described properties that aid in designing software systems or simulations to gain speed using parallel processors. The principle property is that of the independence of operations of a physical system. As described above, one must consider both spatial as well as temporal independence when simulation clocks on different processors are not synchronized. As described in the next chapter, if these independent operations can be modeled using a state-space framework, then the state vectors separating the independent transformations, and the transformations themselves, can be built as software modules or models of a physical system. When satisfying the property of independence defined above, these models or modules can be run concurrently on parallel processors.

CHAPTER 7

MODELS AND SPACES OF SYSTEMS AND SOFTWARE

SOFTWARE SPACES FOR PARALLEL PROCESSING

Expansion of mathematics is motivated by the expanding needs of science and engineering. Driven by such needs, mathematical notation progressed from integers and fractions, to complex numbers and vectors. In the 1950s, engineers developed the *State Space* framework to describe complex control systems in the continuous or discrete time domains, see [7], [60], [128] or [156]. In each case, extension was developed to simplify solutions of complex problems. The underlying goal is to find the best coordinate system or *space* in which to represent a problem. Given the correct choice, problem solutions are substantially simplified.

As problems become more complex, spaces required to simplify their solutions also become more complex. State vectors with hundreds of variables have become common. In addition, a state space may be organized hierarchically to further simplify understanding. For example, one may have subvectors describing position, orientation, velocity, acceleration, etc., where each subvector contains multiple components. The benefits of organizing a space hierarchically - to simplify its use and understanding - is apparent. Hierarchies have been applied to control large complex organizations for thousands of years.

As shown below, software spaces are best mapped into hierarchical databases. Using this concept, we introduce *Generalized State Space* as the mathematical framework to support the solution to complex software problems. Databases used to solve complex software problems are *Generalized Spaces* or *State Vectors*. Software algorithms are *Generalized Transformations* on these spaces. Design of the spaces (databases) is key to simplifying the design of corresponding transformations (algorithms) performed by software, including their future enhancement as requirements grow. Just as in mathematical approaches, one must define the state vectors of a system before defining the transformations that describe the dynamics. This requires expanding mathematical notation beyond that of numbers. The approach described here was first implemented by one of the authors in 1982 when designing the General Simulation System (GSS), a CAD system for designing discrete event simulations to run on parallel processors.

As we expand the complexity of mathematical spaces to characterize complex systems, their corresponding laws and transformations continue to apply, and their interpretations are extended to be more general. Most important, each step enhances the ability to deal with increasing complexity. Our interest is simplifying the design of both parallel processing software and hardware. We use examples to demonstrate these effects.

Frameworks For Representing Complex Dynamic Systems

The position of a body in space can be described in various coordinate systems. Problems of dynamic motion are solved more easily if one selects the “best” space. Best is measured by ease with which problems are solved (a practical measure is the speed of solving test problems in a course on partial differential equations). The classic example is a particle in spherical orbit. It is relatively easy to describe its motion in spherical coordinates, but the complexity increases considerably using Cartesian coordinates. In addition to simplifying the representation of a system, a good choice of *state variables* can make computations much faster.

As indicated above, various coordinate systems can be used to solve problems. However, they are solved more easily if we select the right *space*. Selection of the most convenient space is typically taught in courses in linear systems or differential equations under the topic of *separation of variables*. Separation can be used if the variables form a linearly *independent* set. The property of independence can be verified using well-known tests. The concept of choosing the best coordinate system (state vector) and the property of independence are important principles that can be applied to reduce the effective level of complexity with which one must deal. These concepts apply directly to simplifying software development for parallel processors.

In the early 1960's, electronic circuit designers developed automated tools for solving complex systems of nonlinear differential equations required to simulate electrical circuits. These Computer-Aided Design (CAD) tools allow engineers to describe networks graphically and write equations describing nonlinear components. Programming skills are unnecessary. The code required to run simulations of large complex networks is generated automatically. This provides a huge improvement in design productivity (from months to hours).

For large networks, the number of state variables may be in hundreds, even thousands. Solving design problems may involve optimization runs using hundreds of simulations. Each simulation may involve hundreds of nonlinear differential equations. Speed and accuracy are the driving forces in designing these CAD systems. If a computer takes days to produce a design, only a few test points are produced in a week. This provides substantial motivation to drastically cut the time to obtain a solution, leading to special approaches to gain multipliers on speed.

Electrical Network Analogies

Electrical engineers have evolved a graphical representation for models of complex electrical networks using interconnected icons of basic element types: resistors, capacitors, inductors, generators, etc. As illustrated in Figure 7-1, these elements can be used to build up hierarchical models of higher order elements, e.g., transistors, transmission lines, etc. Such a drawing defines the differential equations that describe the dynamic changes in electrical voltages and currents in the circuit. A vector-matrix representation is shown in Figure 7-2 using the *state-space framework*, where X represents a vector of state variables and U represents a vector of external driving forces. Given the initial conditions, the state of the circuit is defined for all time thereafter. In other words, the dynamic behavior of the network is defined by the symbolic network architecture.

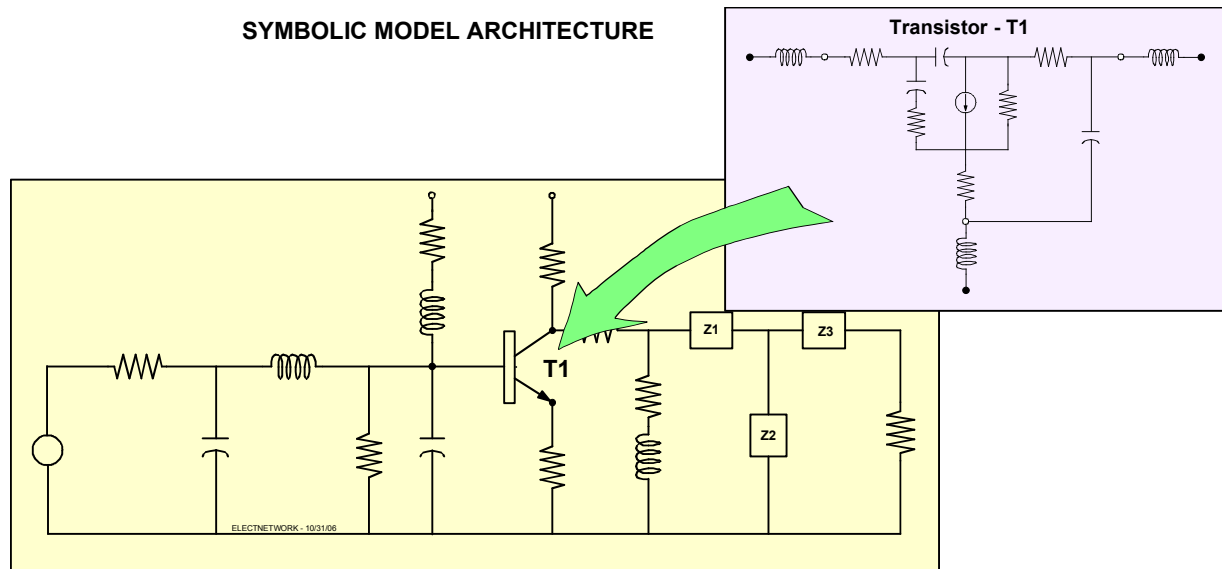


Figure 7-1. Architectural representation of an electrical network.

$$\begin{bmatrix} X(T+\tau) \end{bmatrix} = \begin{bmatrix} A(T) \end{bmatrix} \cdot \begin{bmatrix} X(T) \end{bmatrix} + \begin{bmatrix} B(T) \end{bmatrix} \cdot \begin{bmatrix} U(T) \end{bmatrix}$$

Figure 7-2. Mathematical (State Space) Representation of an electrical network.

Requirement Specification And Design Specification

Designers of electronic circuits start with functional requirement specifications and produce drawings and design specification documents, see Figure 7-3, describing the network to be fabricated in production. Negotiation of changes in functional requirements and testing of proposed fabrication processes for reliability are a significant part of the engineering design process. Engineering judgment, supported by mathematical calculations, provides a critical part of this process. Testing is used to validate models of the nominal behavior of a network as well as variations in component values due to production and environmental changes. Ensuring reliability requirements are met in anticipated “worst case” environments is a hard constraint.

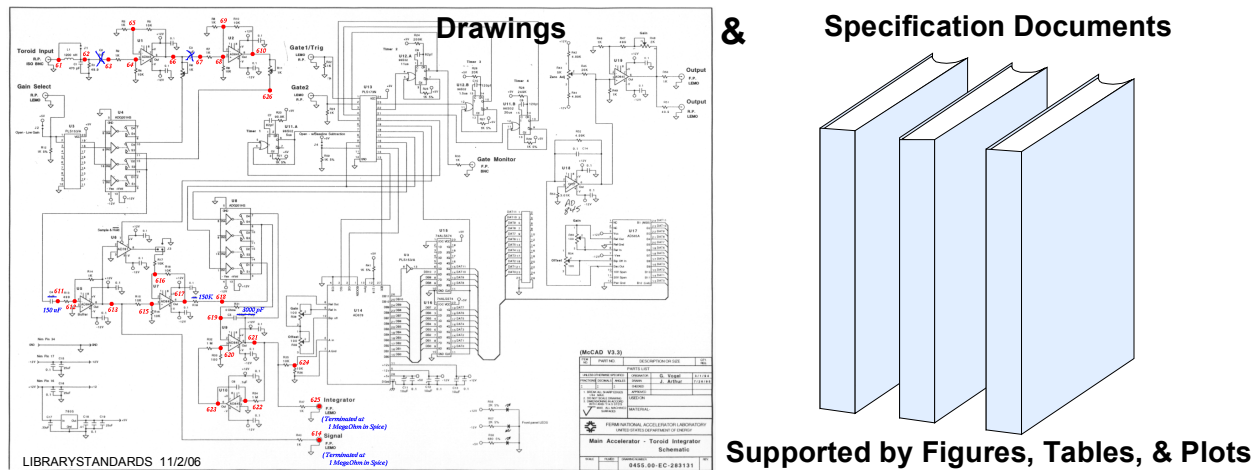


Figure 7-3. Requirements and design specification documents.

Use Of Electrical Analogies For Enhanced Productivity

Because of the development of a complete and consistent theory for electrical network design, electrical analogies are used in many other fields, such as those involving mechanical design, fluid flow, etc. Additionally, characterization of nonlinear components and development of fast and accurate solutions of the differential equations representing these networks provides a highly productive facility, minimizing the burden of analysis of different designs. These facilities have been incorporated into numerous Computer-Aided Design (CAD) tools that have reduced the time and resources required to build simulations and complete a complex design by orders of magnitude. Obtaining fast and accurate solutions to posed design problems provides huge improvements in productivity in the fabrication stage as well as in production of the design specifications.

APPLICATION TO DISCRETE EVENT SIMULATION AND SOFTWARE

There are four reasons why *discrete event* simulation is a good software analogy and starting point. First, an electrical network analogy for this type of simulation is easy to derive. Second, it has been shown, [36], that discrete event simulation can be used to solve highly nonlinear problems in the time domain that are otherwise intractable using differential or discrete time equations.

Third, this same reference provides examples of more easily understood solutions even for linear problems using a discrete event CAD system versus using differential or discrete time equations. This is because of the higher degrees of abstraction imposed when using standard mathematical frameworks. After reviewing sufficient examples, it becomes clear that such abstractions make it much more difficult to model realistic behavior in a way that is easily understood.

Finally, software is a subset of discrete event simulation when using the General Simulation System (GSS), see [67]. This system, designed in 1982, is based upon the State Space framework. We will start with discrete event simulation using GSS and State Space as it is used to represent communication as well as electrical networks, and show how this form of simulation is a generalization of software.

GENERALIZED STATE SPACE

Software modularity was considered in the late 1960s and early 1970s, see for example Gauthier and Pont, [59], and Parnas, [107]. However, the concepts lacked precise definitions, and were soon overtaken by the use of C-based languages (C, C++, C#, Java, etc.) and OOP. A major drawback of the evolution of these later approaches is the use of abstractions and data hiding. These properties obscure what routines (instructions) share what data. The use of inheritance exacerbates this obscurity. As shown below, such approaches inhibit the understanding and determination of the property of independence - the key to software architecture. This, coupled with the terse and cryptic nature of C-based languages, makes it difficult to control growing complexity in large software systems - independent of using parallel processors.

The approach described here adopts Generalized State Space as the framework for defining software architectures. It was devised originally to define the GSS discrete event simulation environment and is somewhat different from the approaches described by Gordon, see [63] thru [65]. Whereas State Space is known to provide a convenient mathematical framework for representing most any type of dynamic system, see [60], [128], and [156], Generalized State Space extends this facility to incorporate general decision algorithms into the limited mathematical framework. This is particularly useful in simplifying the representation of highly nonlinear systems. This approach has bred solutions that provide order-of-magnitude reductions in run time on a single processor and simplify the use of parallel processors. It extends the problem solution space beyond current mathematical frameworks, including those known as “discrete event systems” as described in [115] and [116].

The Property of Independence - The Basis For Software Architecture

Development of the Generalized State Space framework was motivated by multiple factors. One was the requirement to embed decision algorithms within mathematical models. For example, communication system designers want to embed natural language conditions within models as shown below.

```
IF MESSAGE_TYPE IS CONTROL, THEN ... ,  
ELSE  
IF MESSAGE_TYPE IS DATA, THEN ... .
```

Additional factors were the lack of simulation scalability and excessive run-times of prior approaches. These factors motivated the development of a complete and consistent state space definition of discrete event simulation. A summary of this is provided in *Simulation Of Complex Systems*, see [36]. In that reference, the Generalized State Space framework is compared to more common mathematical formulations of state space. We note that much of the theoretical framework was developed in 1982 as part of the development of the General Simulation System (GSS), a product that has been in the international market since 1984, see [73], [127] and [157].

Prior to the technology embodied in GSS, simulations of large mobile communication systems required 5 to 7 days to run a 2 hour scenario. This led to a major GSS design requirement - simulations must run on a parallel machine. This implies that two or more processes[†] must be able to run concurrently on separate processors. This requires that the concurrent processes must be *independent*. As clarified below, independence implies that the processes share no data. This led to the decision to separate data from instructions so that the independence property could be established and tracked in the development environment.

The original design called for a connectivity matrix, described below, showing which processes share what data. Then when allocating processors to processes, the connectivity matrix could be used to determine if a process may run concurrently with those already running.

Software Architecture - Modularity & Independence

In engineering, breaking complex systems into independent modules is embodied in the architecture, a concept that has been misunderstood in software. This is because *architecture describes connectivity*, i.e., how a module is connected to other modules. *Engineering architectures represent the time-invariant properties of a system - not flow of control* (they are not flow charts). Creating software to take maximum advantage of a parallel processor requires that the software be decomposed into independent modules that can run concurrently. This architectural decomposition must reflect the inherent parallelism of the system it represents.

Descriptions of architecture are not convenient using algebraic or linguistic representations. Like other engineering fields, software architecture is best described with drawings, depicting how modules are connected. Only then can one visually observe independence - the key property supporting concurrency. Flow charts - or graphical variations on flow charts - are of little use when describing the property of independence.

The Separation Principle

A software development environment for parallel processors must support a designer faced with creating an architecture of independent modules based upon knowledge of the inherent parallelism in a system. To accomplish this, one must *separate data from instructions at the design interface* level so that module independence (or lack of it) is clearly recognized. Defined in 1982 in the design of the General Simulation System (GSS), this has become known as the *Separation Principle*, [80]. As described below, Generalized State Space was devised as the mathematical framework for implementing GSS.

[†] *Process* as used here is similar to an assembler language subroutine that contains only instructions that reference only external data (by pointer). It has no relation to a UNIX process.

The Generalized State Space Framework

The Generalized State Space framework capitalizes upon the concepts of State Space developed for control theory by extending the mathematical definitions of vectors and transformations. We start with the concept of a *Generalized State Vector*. Instead of restricting a vector to numbers, it can take on states described by words. For example, the state LIGHT may take on the values RED, YELLOW, or GREEN. In addition, transformations on a state need not be restricted to mathematical operations. For example, we may want to say IF LIGHT IS YELLOW, SET LIGHT TO RED, a *Generalized Transformation*. Given this facility, one may view a computer program as consisting of generalized state vectors (data) and transformations (instructions).

Using the Generalized State Space framework, the Separation Principle is achieved by storing all data in *resources* that generally contain hierarchical data structures (refer to Figure 5-2 in Chapter 5). Resources are depicted as ovals in architectural drawings as illustrated in Figure 7-4. *Processes*, as defined above, contain instructions in the form of hierarchical rule sets (refer to Figure 5-3 in Chapter 5). They are depicted as rectangles.

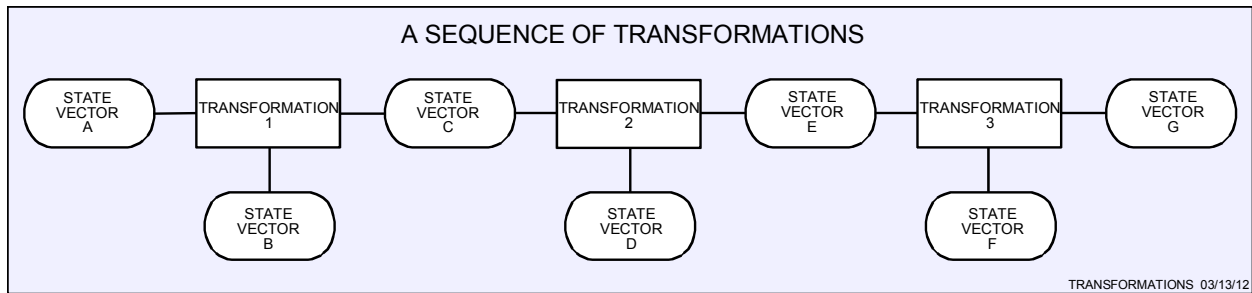


Figure 7-4. State vectors and transformations.

In Figure 7-4, each transformation has a dedicated state vector and a shared state vector. Transformation-1 has state vector A as input, has state vector B for dedicated use, and shares state vector C with transformation 2. Therefore, transformations 1 and 2 are not *independent*.

As used here, the property of *independence* ensures that processes running on a parallel processor produce *complete and consistent* results for a given set of initial conditions. Consider that state vectors C, D, and E have initial values C_i , D_i , and E_i . When run on a single processor (sequential machine), transformation 2 will produce the same outputs: C_o , D_o , and E_o for a given set of inputs every time it runs; i.e., the results will be *complete and consistent*. If while it is running, one of the resources is changed from the outside, the results may not be complete and consistent. This is because the data being accessed is not *consistent* relative to transformation 2.

If transformations 1 and 2 run concurrently, shared state vector C could be changed by either, rendering the data as recognized by the other as *potentially inconsistent*. Therefore, in general, they cannot operate concurrently. Similarly, transformation 2 is directly coupled to transformation 3 (by shared state vector E), is not independent of it, and thus cannot run concurrently with it.

However, transformations 1 and 3 can operate concurrently since they share no state vector directly and are therefore independent. Transformation 2 can operate only when transformations 1 and 3 are both idle.

The critical property determining complete and consistent results of the transformations, and thus inherent parallelism, is that of independence. This implies that *transformations are spatially independent when they share no state information*. The separation of state information (data) from transformations (instructions) provides the basis for the Separation Principle for software as described in [35], [37], and [41]. The Separation Principle yields drawings of software architecture, similar to that in Figure 7-4, providing direct visualization of model/module independence. We will show below that this property - two processes not connected to the same resource - is one of *spatial independence of the processes*.

Discrete Event Simulation As A Generalization Of Software

As stated above, discrete event simulation models may be characterized in terms of generalized state vectors and transformations. This leads to a visualization of the connectivity properties of transformations as shown in Figure 7-4. In this figure, generalized state vectors contain hierarchical data structures. The generalized transformations contain hierarchical rule structures, where a rule contains high level language instructions that implement complex algorithms. These are general software facilities described in succeeding chapters.

Parallelism, Architecture, and Decomposition

Theoretically, one can determine the inherent parallelism in a system by creating a model whose elements are represented by state vectors and transformations similar to that in Figure 7-4. As shown in [40], a proper choice of state vectors will maximize the independence of models, taking maximum advantage of the inherent parallelism, and optimizing concurrency. Various authors have shown that any physical system may be modeled to within any measurable degree of accuracy using the state-space framework, see for example [128] and [156]. From this we may conclude that the inherent parallelism in a system may be modeled by representing its operations using the Generalized State-Space framework. These models may be mathematical models, discrete event simulation models or general pieces of software.

Using this concept, software functions are considered transformations on data. Functional requirements may be translated into a set of vectors and transformations. Mathematically, transformations may be performed using different vector spaces, some of which are more efficient than others. Using the common example, motion of a particle on a spherical surface is most easily described in a spherical space. When modeling complex systems in software, it is important to understand the different coordinate systems (state vectors) and transformations that are best used to implement different functions.

These concepts are significant when translating functional requirements into software. When devising mathematical transformations, selecting the best set of independent state vectors is a critical part of finding the best solution. “Best” usually translates to speed which, in turn, translates into minimum operation counts, see Hachtel et al, [71]. This concept applies directly to software design, independent of parallel processing. Finding the best set of state vectors can make a huge difference in transformational burden. When dealing with complex data sets, design of the tables used for transformations of data is usually key to simplifying both the approach, resulting code, and speed of operation.

When striving to take advantage of the inherent parallelism in a system, one must determine the best architecture of software modules that can run concurrently on a parallel processor. Picking the best set of state vectors is key to solving this problem. Again, best translates to run-time speed and simplicity of transformations.

State Space Representation Of GSS

Within a GSS simulation, a model only has access to a subset of the simulation state vector when a process in that model is running. We will call this the *model state vector*. A subset of the model state vector contains those resources that are contained within the model. Because this GSS state vector may contain character as well as numeric data, it is called a *generalized state vector*. The state space representation of a GSS model is shown in Figure 7-5.

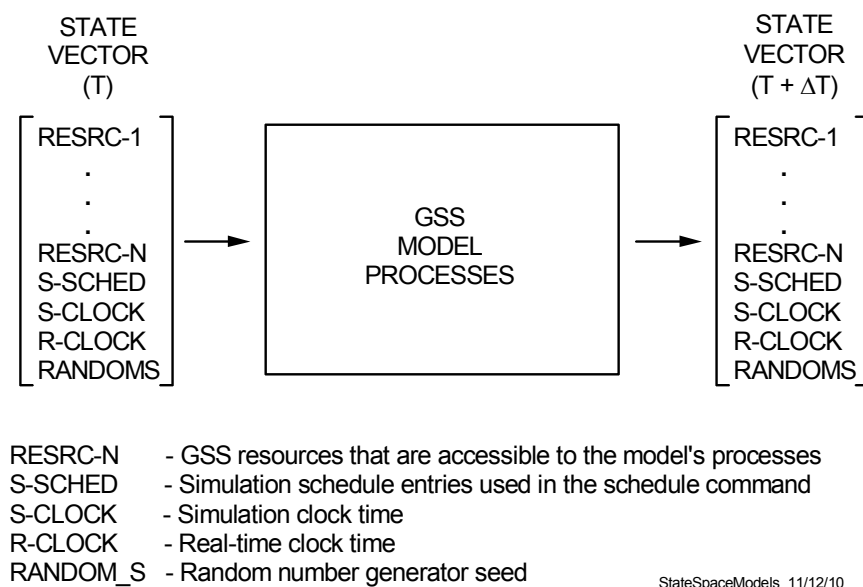


Figure 7-5. State space representation of GSS.

The state vector of a GSS model consists of the following items and their corresponding information elements:

- ACCESSIBLE RESOURCES - The information contained in resources to which processes within the model are attached. Note: Shared resources may or may not reside within the model.
- SIMULATION SCHEDULE - An interface to the scheduler.
- SIMULATION CLOCK - The time of the simulation clock, including priority, including when it schedules another process.
- REAL TIME CLOCK - The value of the real-time clock if it is used.
- RANDOM NUMBER GENERATOR - The current value of random number generator seeds if they are used.

A GSS process is scheduled based upon the logic within itself or in other processes. When a GSS process *runs*, it may schedule itself or other processes at specified times in the future, or at the current time. GSS processes run in zero *simulated* time. The simulation clock advances based upon the next scheduled process. At any time, the state of a model depends solely upon its state vector. When a process in a model runs, its *terminal state*, i.e., the value of its substate vector - when it passes control back to GSS - depends solely upon its *initial state*, i.e., the initial value of its substate vector, and the rules within the process.

When processes in another model share a part of the state vector of a given model, then any future state of the given model is, in general, dependent upon the rules in the other model, since they can change the given model's state vector at a different instance in time.

ANALOGY TO SYMBOLIC MODELS USING STATE SPACE

The state space representation of a GSS model, Figure 7-5, is analogous to a set of differential equations that represent the state of a dynamic system at any instant in time. All future states are represented by the *equations of motion* in state space notation, and the initial conditions, reference Schweppe, [128].

In GSS, the interconnection of resources and processes, see Figure 7-4, is analogous to the electrical circuit drawing in Figure 7-1. Each has its corresponding *rules* and *storage* underlying each primitive element. In the case of electrical circuits, there are constituent equations that describe the changes in energy storage in differential form for each primitive icon. Representation of any system element must conform to this form of change.

In the case of GSS, sets of rules operate on sets of attributes contained in data structures to define the elementary change relationships in a model. Using GSS, the model shown in Figure 7-4, along with the underlying rule and data structures define the total state of the simulation at any point in time after the initial conditions in *generalized state space*.

Choosing the Most Convenient Reference Frame

As implemented in GSS and described above, the generalized state space framework supports the representation of discrete event systems as well as discrete and continuous time systems. Figure 7-6 illustrates the generalized state space as providing an underlying framework for representing dynamic systems.

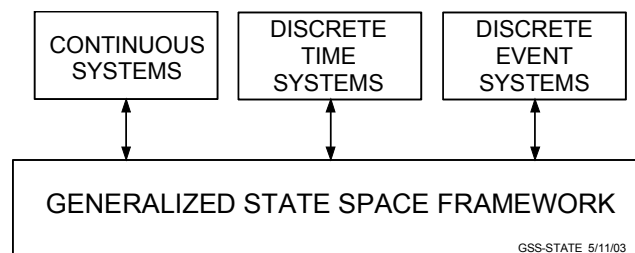


Figure 7-6. Generalized State Space:
- an underlying framework for representing dynamic systems.

The difference between model representations of a system's dynamics may translate into huge differences in productivity. A particular representation can be selected to make it easier to analyze or predict specific system behavior. If a system is conveniently represented by a set of differential or difference equations, then one of those representations may be best. If the system is more easily described by sets of rules operating on sets of attributes that may contain nonnumeric elements, then that representation should be chosen.

Since the advent of the digital computer, people have moved from analytical methods for integrating differential equations to heuristic algorithmic methods, especially when the systems represented are either nonlinear or nonstationary. Fast numerical algorithms for solving stiff nonlinear systems typically use complex heuristic approaches, see [70]. All of these approaches can be implemented easily using GSS rule and attribute structures. As computers provide significantly greater memory and corresponding speed advantages, the space for solving problems is growing, alleviating restrictions to abstract numerical methods for solution, and allowing rapid movement toward heuristic rule-oriented approaches using complex data structures. These approaches are compared in *Simulation Of Complex Systems*, [36].

We note that GSS is a discrete event simulation environment, where flow of control may depend upon a huge set of event strings that in turn depends upon a huge state space. Although sequences of events may be deterministic, they are virtually unpredictable in a large simulation. When selecting frameworks for solving such problems, one must ensure completeness and consistency so that, depending upon their inputs, solutions converge to the expected outputs unambiguously. When developing GSS, the State Space framework was used to ensure these properties.

Having selected GSS as the overall framework, the analogy then becomes one of selecting the best set of information vectors (GSS Resources) to represent the system attributes. Depending upon how the resources are designed and structured, the rules (GSS Processes) may be much simpler to understand, build, and modify. This is determined by the *independence properties of the architecture*, i.e. the interconnection of resources and processes - *not the code!*

Mapping Into Software

Since there are no language statements in GSS that are specific to simulation, although the SCHEDULE and CANCEL statements may be used as such, the language is a complete and eloquent parallel processor software language. It is a very rich language that is used for the GSS counterpart, the Visual Software Environment (VSE). In VSE, the SCHEDULE statement is used to invoke processes on different parallel processors. All of the architectural properties available to GSS users also reside in VSE. The fact that GSS is written in VSE implies that VSE can also be used to build simulations.

SOFTWARE ARCHITECTURE

As illustrated in Figure 7-4, software architects can decompose a system into modules by grouping resources and processes into an *elementary module*. *Hierarchical modules* are created by grouping modules into higher level modules. Figure 7-7 shows a library module that is sufficiently complex to warrant its own drawing. In general, modules are independent if they share no resources (i.e., they are not connected). Having developed an architecture, developers can implement the data structures and rules using the resource and process languages. These may be edited directly on the drawing as illustrated in Figure 7-8. The languages do not permit the declaration of scope rules. It is the architecture that determines the sharing of data and corresponding independence of modules.

Unless one has witnessed directly the development of such architectures, the above discussion may take time to comprehend. Having used it, it is apparent that architecture as defined here is as critical to software design as it is to any other engineering discipline, with or without parallel processing. It is why productivity multipliers are very high when using this CAD environment, especially in the support mode when a new person has to understand what another has built.

Taking Advantage Of Architectural Information At Run-Time

To take advantage of a parallel processor at run-time, the OS must map threads onto processors to maximize the speed multiplier. A programmer faced with generating complex algorithms should not be concerned with this problem. Similarly, a compiler will have little success trying to interpret an architect's decomposition of modules from the code. Finally, the operating system will not be very successful in determining where to map threads based upon current run-time statistics, especially if they are nonstationary - as they are in most discrete event simulations due to huge nonlinearities.

If there is sufficient inherent parallelism in the system, architects can decompose the software into large Independent (IND) modules. As described below, threads are contained within IND modules. Because threads in one IND module are independent of those in another, they can run concurrently on separate processors without concern for synchronization.

The architectural information that characterizes inherent parallelism is contained in databases that support the CAD development environment. A Run-Time System is generated from that information to control OS calls that allocate processors to modules. It also ensures that the resources (data) reside with the processes (instructions) that use them. If there are enough processors to house the IND modules, load balancing (migration) is unnecessary, see [44]. Synchronization of data accesses between IND modules is handled automatically as described in later chapters.

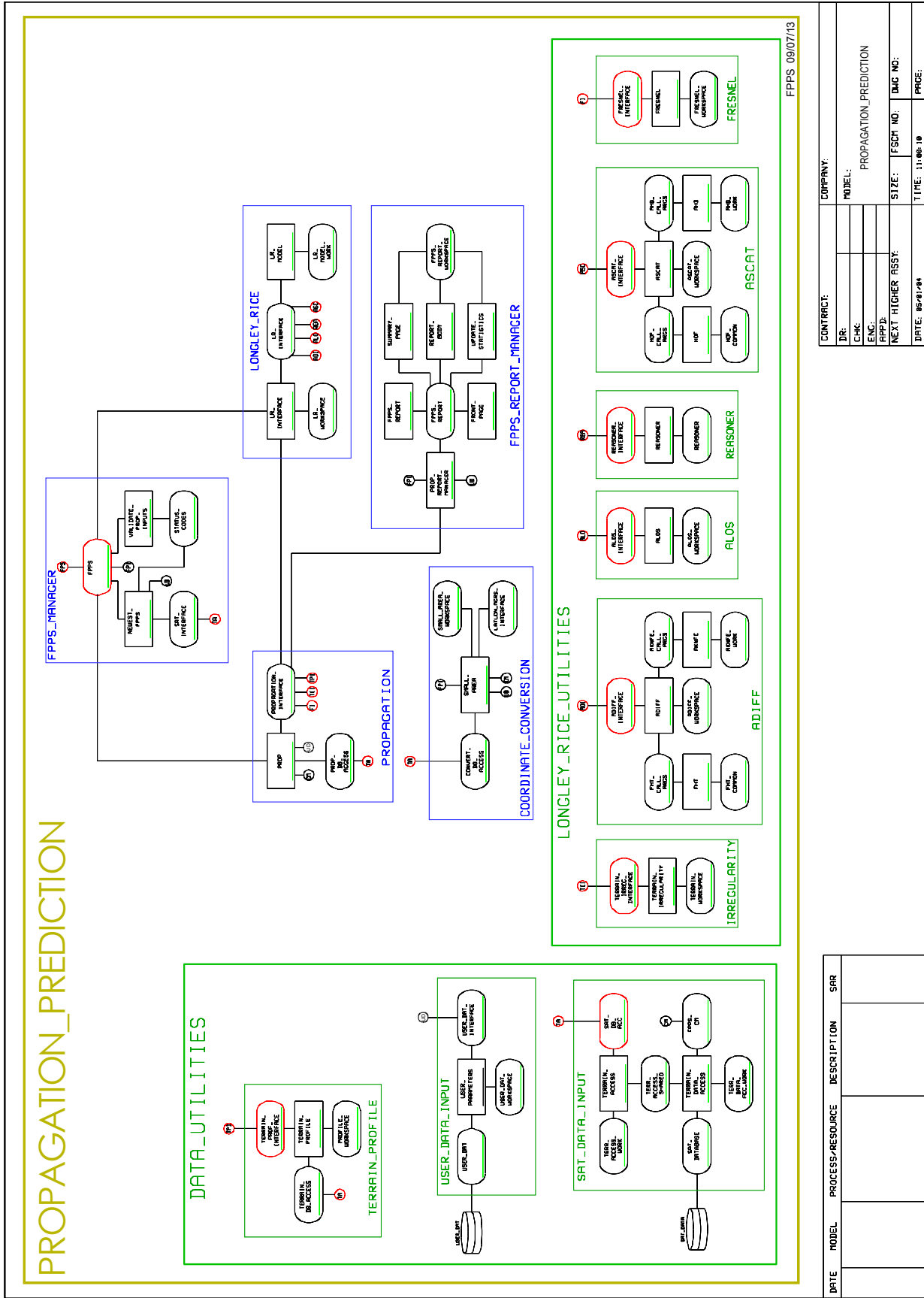


Figure 7-7. Engineering drawing of a library module.

Scalability

Increasing complexity may cause a software development effort to scale nonlinearly, i.e., the effort required to build a system increases faster than its complexity. This is characterized by Fred Brooks in *The Mythical Man-Month*, [19]. However, linearity depends upon independence. Linear scaling can be achieved by maximizing module independence. This phenomenon is the result of the separation principle, see [41].

As indicated above, current languages provide limited control over the design of large complex software systems. Architects of industrial buildings, ships or airplanes would find it very difficult, if not intractable, to produce large complex designs without drawings. It is now apparent that software is no different. Having used the CAD interface and drawings to produce architectures, and having observed module independence by visual inspection, it becomes obvious that building software without drawings is no different from any engineering field that depends upon drawings. Visualization of the architecture is critical to understanding - and controlling - increasing complexity when building and supporting software.

SOFTWARE LANGUAGE PROPERTIES

When considering a programming language, one is concerned with two issues: (1) the speed with which a high quality product can be built and enhanced; and (2) the speed with which it runs, see [2]. These factors are key to building large simulations. When assessing the validity of a model, subject area experts must be able to easily understand the algorithms as well as the architecture. This depends directly upon understandability of the language used to describe the algorithms. *Understandability* is a measure of the relative ease with which others (including subject area experts) can understand an algorithm. This directly affects the effort required to validate a model as well as build and debug software. Having separate language translators for data and instructions helps one focus on the understandability of each.

Figure 7-9 illustrates the problem of developing software systems and simulations. Today, applications run on processors with separate elements optimized for accessing data memory and instruction memory. Humans can translate their problem into a user-friendly language (software space). As shown by history, this language must be designed for human understanding, making it easy for subject-area experts to map their problem into a software space that best suits their applications. Translation of human-oriented languages into binary is done by the computer. With languages designed to simplify mapping applications into software space, the computer translation becomes much more complex. But that is exactly where the burden should be.

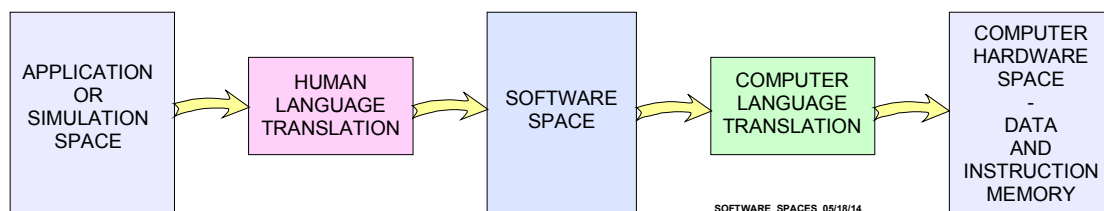


Figure 7-9. The environment of software language translators.

Factors affecting both speed and understandability of a complex algorithm include the manner in which the state vectors are structured. Hierarchical structures are used historically to control large complex organizations where speed and precision of operations are important. This property is illustrated in the resource shown in Figure 7-10. Hierarchical data structures, organized around the application, *not the data type*, greatly enhance understandability. It allows them to be grouped to support meaningful architectures as illustrated in Figure 7-7.

RESOURCE NAME: MESSAGE_FORMATS			
MESSAGE			
1	SYNC_CODE	CHARACTER 6	
	ALIAS VALID	VALUE '101010',	
		'010101'	
1	TYPE	STATUS FORMAT_A	
		FORMAT_B	
1	CONTENT	CHARACTER 46	
FORMAT_A REDEFINES MESSAGE			
1	PAD	CHARACTER 14	
1	HEADER		
	2 PRIORITY_A	STATUS FLASH	
		IMMEDIATE	
		ROUTINE	
	2 ORIGIN_A	INDEX	
	2 DESTINATION	INDEX	
	ALIAS BROADCAST	VALUE 0	
1	BODY		
	2 LENGTH	INTEGER	
1	TRAILER		
	2 MESSAGE_NUMBER	INTEGER	
	2 TIME_SENT	REAL	
	2 TIME_RECEIVED	REAL	
	2 ACKNOWLEDGMENT	STATUS RECEIVED	
		NOT_RECEIVED	
	2 LAST_SYMBOL	CHARACTER 2	
	ALIAS TERMINATOR	VALUE '\\', '/', '<<', '>>'	
FORMAT_B REDEFINES MESSAGE			
1	PAD	CHARACTER 14	
1	HEADER		
	2 SOURCE	INDEX	
	2 SINK	INDEX	
1	BODY		
	2 CONTENTS	CHARACTER 42	

Figure 7-10. Example of a hierarchically structured state vector (Resource).

Hierarchical data structures also support group moves, e.g., moving a large character string into the MESSAGE structure in Figure 7-10. This simplifies the algorithm, substantially improving speed as well as understanding. Many other features in the resource language, e.g., the STATUS and ALIAS attributes used in Figure 7-10, contribute to enhanced understandability of a process. This is apparent in the example that follows.

Processes are transformations that contain hierarchical sets of rules, an example of which is shown in Figure 7-11. Data cannot be declared in a process. Processes may only reference data defined in resources to which they are connected (by a line in the architecture). Resources connected to a process are referenced automatically by pointer. With this paradigm, programmers are only concerned with indexing, not memory management (ultimately controlled by the OS).

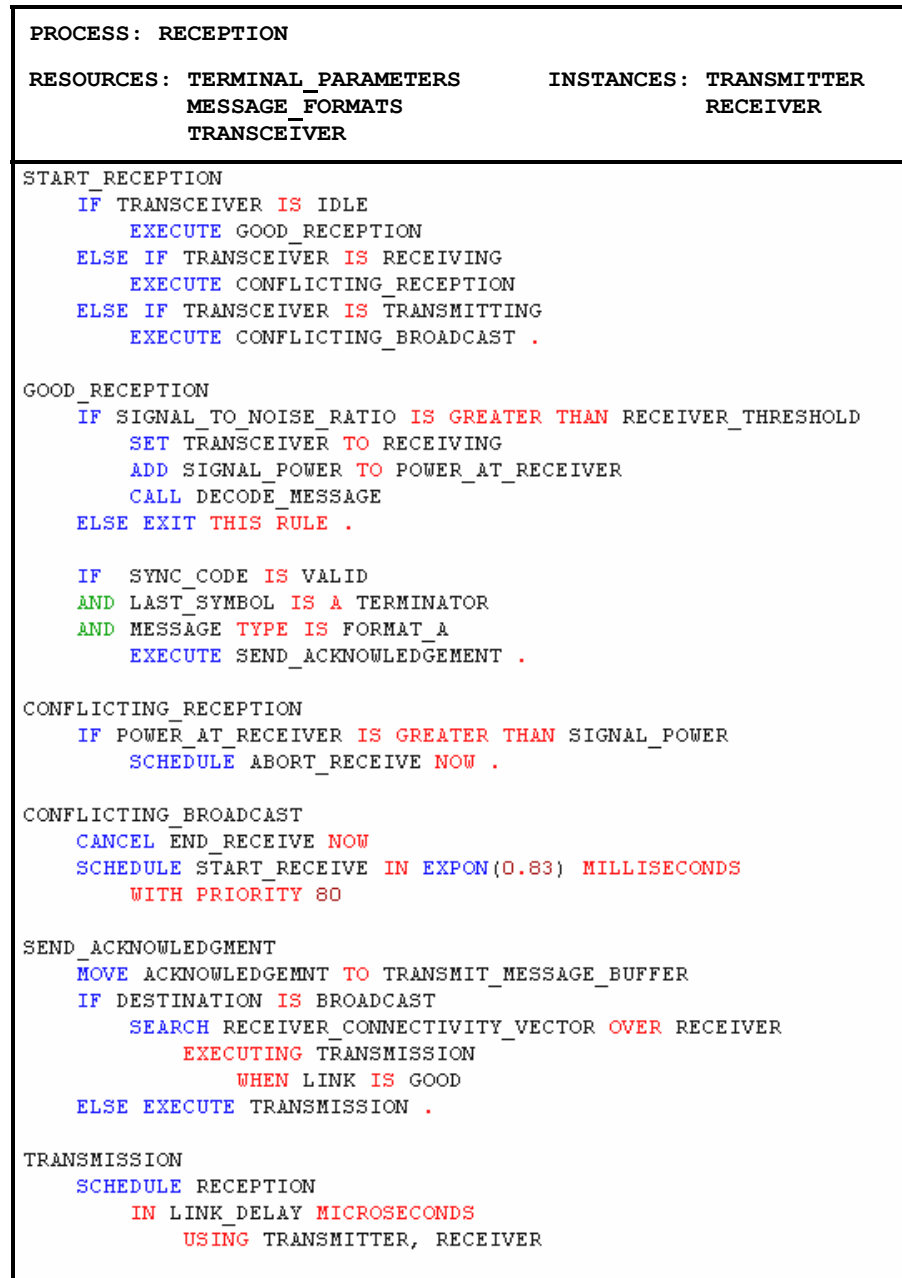


Figure 7-11. Example of a hierarchically structured transformation (Process).

Rules (labeled in column 1 in Figure 7-11) implement one-in one-out control structures as advocated by Mills, [102]. They are invoked by the EXECUTE statement. After their execution, control is returned to the statement directly following the EXECUTE. This provides a hierarchy of control within a process that simplifies understandability. The result is that complex conditional statements may be reduced to their minimum hierarchies, eliminating nested IF statements and removing assignment type statements from within conditional control structures. This makes complex control structures much easier to understand. Rules may be invoked from multiple EXECUTE statements (the language contains no GO TOs).

Speed is the major driving force in simulation language design. As indicated above, hierarchical data structures support group moves that can contribute order-of-magnitude improvements in run-time speed. In Figure 7-10, MESSAGE is *redefined* by FORMAT_A and FORMAT_B; these are templates over the same area of memory. Individual data attributes can be moved in large groups as long as overall sizes of the structures are matched. However, the REDEFINES statement eliminates unnecessary MOVES. All of the fields are filled in one move (one instruction fetch). This requires memory to be mapped WYSIWYG (What You See Is What You Get), with no “word-boundary alignment” (data has not been mapped as words since the late 1960s).

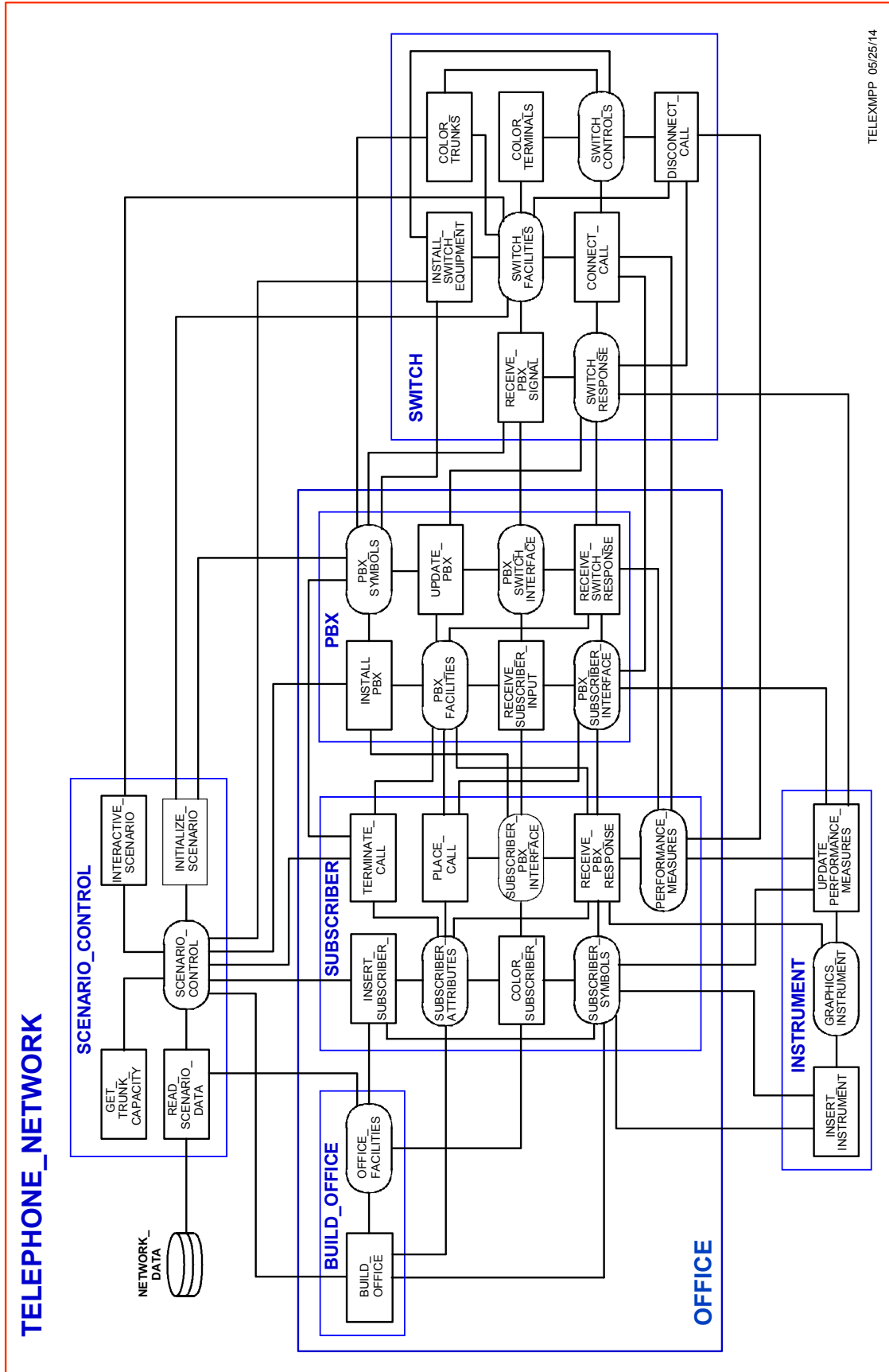
In state space, a major factor affecting speed and understandability of a complex transformation is selection of the state variables. In software, this is equivalent to the design of the underlying data structures supporting an algorithm. In the prior example, message processing is simple. One need only move a message string into MESSAGE. All of the fields and conditions are immediately available in the data structure to process FORMAT_A or FORMAT_B messages. As already indicated, speed is enhanced by the same simplification. These facilities depend upon the ability to easily create and use complex hierarchical data structures.

From Figures 7-10 and 7-11, one can see that the languages are designed for speed and understandability. This supports ease of change by other than the original author, and validation by subject area experts (non-programmers). Terse languages, that infer complex meanings with minimum keystrokes, risk misunderstanding. The reasons follow from information theory where redundancy is shown to improve real understanding, see [130].

A frequent misperception is that verboseness, a form of redundancy, implies loss of speed. In fact, there is no relationship between speed of execution and verboseness of the source code. The translator for a natural language may be very complex, but that is exactly where the burden should be placed; alternatively, it is on the person trying to understand the algorithm.

VISUALIZATION OF SOFTWARE ARCHITECTURE

Geometry and algebra each play important roles in engineering. Theoretically, one could do away with the images provided by geometry. In practice, those who can use geometry have a significant advantage. Figure 7-12 illustrates a model that was developed using a hierarchy of models, but without the use of engineering drawings (the drawings were done after the fact). As is typical in conventional software, data is shared everywhere, with no visible check on independence.



TELEXMPP 05/25/14

Figure 7-12. Telephone network model - a pre-drawing version.

Figure 7-13 shows the same application developed using the VisiSoft CAD environment, see [67]. This example demonstrates the importance of visualization of the independence properties of software. But now take away the Separation Principle with the ability to easily group large data structures. If the shared data is scattered, as when using a C-based language, the drawing becomes intractable. Note also the Instanced Modules, OFFICE(20) and SUBSCRIBER(50). Each instance of OFFICE is independent (no resources shared directly).

Another critical property of this approach is the *Connectivity Matrix* shown in Figure 7-14. The blue boxes indicate the module boundaries for elementary as well as hierarchical modules. For each resource in Figure 7-14, one can see the processes connected to it (denoted by an X) to share the data. The connectivity matrix defines process and module independence. The more sparse the matrix, the greater the independence of modules, and the more simple the transformations, a concept taught to engineers in linear system theory. This information is maintained by the development environment and passed to the run-time environment to determine what processes/modules can run concurrently on parallel processors.

In Figure 7-14, the SCENARIO_CONTROL module shares the SCENARIO_CONTROL resource with processes in each OFFICE module instance (R - Read Only) and the SWITCH module (R), being temporally independent. However, this is only during initialization making those modules effectively independent after initialization. The OFFICE module also shares PBX_SWITCH_INTERFACE and SWITCH_RESPONSE with the SWITCH module (Rs), so those modules are temporally independent throughout the simulation and require minor synchronization, a facility provided automatically by VisiSoft Inter-Processor (IP) Resources (outlined in blue in Figure 7-13). OFFICE also shares PERFORMANCE_MEASURES with INSTRUMENT (R) as a seldom-used temporally independent one-way output. Because IP Resources are automatically copied and synchronized by the tailored Run-Time System (RTS) they can be shared concurrently.

Figure 7-15 illustrates this concept. It contains a model of a digital radio system from a simulation used to support the design of large radio networks on moving platforms (land, sea, air and space). The SAT_COMM_RADIO_SYSTEM model is one of many in that simulation. In this example, it is instanced 300 times, i.e., a network of 300 radio systems can be simulated using this model. Except for two submodules at the bottom, all modules in an instance are independent of those in other instances. This implies that 300 processors can be used to house each instance, and these instances will only exchange information through two submodels at the very bottom of the architecture.

Of 124 processes in an instance, 4 share resources between instances. Processes that share resources between independent modules are controlled automatically by a run-time system that ensures synchronization while minimizing wait times. If an instance is active, many of the levels in the protocol hierarchy may be active. Except for processes sharing interface resources between instances, all other processes (threads) can run concurrently, with those in other instances, with assured independence. The efficiency of processor utilization will depend upon the scenario as well as other factors.

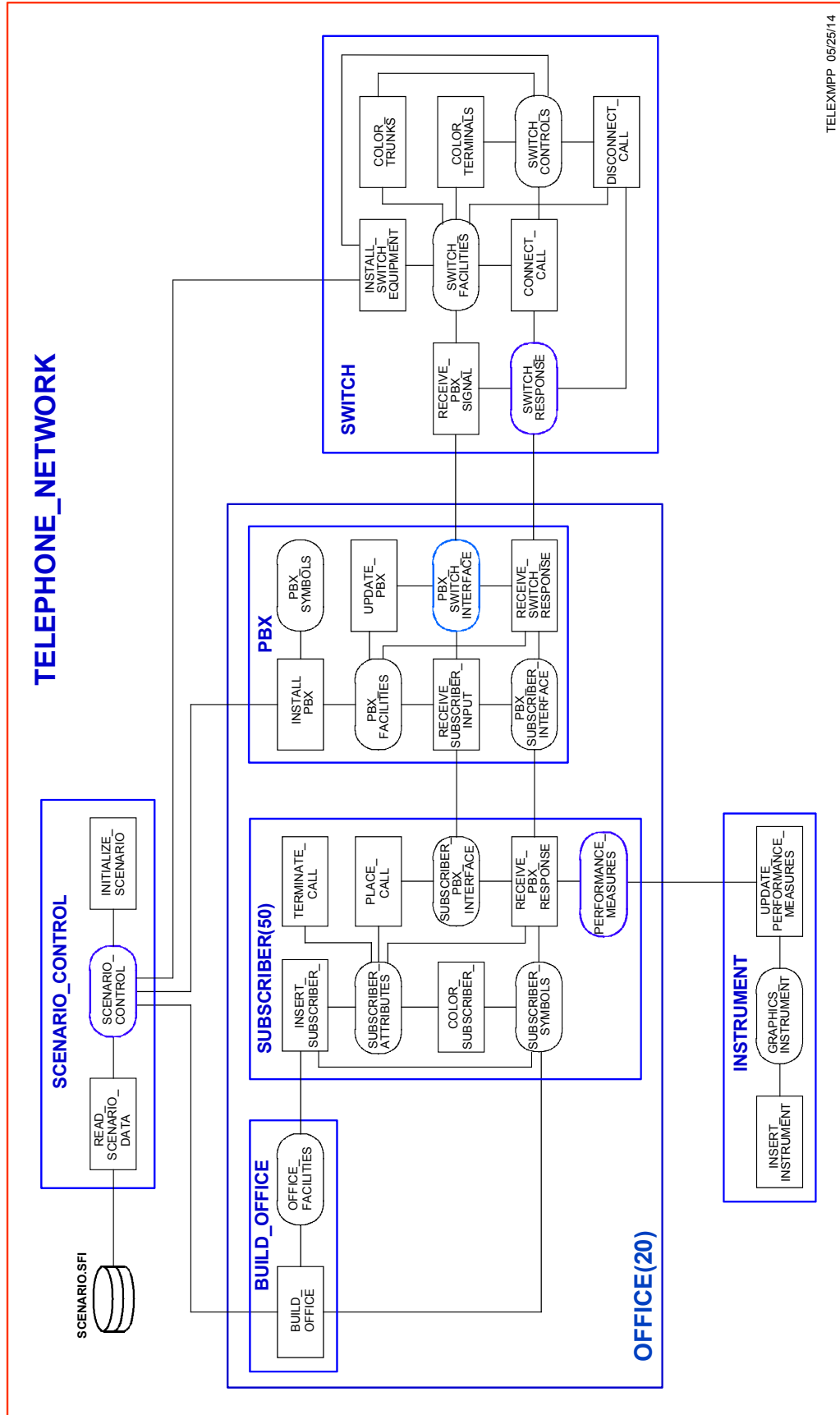


Figure 7-13. INTER_OFFICE_NETWORK Model using RTG.

MODULES	IND MODULES	PROCESSES	RESOURCES	SCENARIO_CONTROL	OFFICE_FACILITIES	SUBSCRIBER_SYMBOLS	PERFORMANCE_MEASURES	SUBSCRIBER_ATTRIBUTES	SUBSCRIBER_PBX_INTERFACE	PBX_SUBSCRIBER_INTERFACE	PBX_FACILITIES	PBX_SYMBOLS	PBX_SWITCH_INTERFACE	SWITCH_RESPONSE	SWITCH_FACILITIES	SWITCH_SYMBOLS	GRAPHICS_INSTRUMENT
READ_SCENARIO_DATA			X														SCENARIO_CONTROL
INITIALIZE_SCENARIO			X														
INTERACTIVE_SCENARIO			X														
GET_TRUNK_CAPACITY			X														
BUILD_OFFICE			R	X	X												OFFICE
INSERT_SUBSCRIBER				X	X												
PLACE_CALL								X	X								
TERMINATE_CALL					X			X									
COLOR_SUBSCRIBER					X			X									
RECEIVE_PBX_RESPONSE				X	X	X	X	X	X								
RECEIVE_SUBSCRIBER_INPUT									X	X	X		X				
INSTALL_PBX			R								X	X					
UPDATE_PBX											X		X				
RECEIVE_SWITCH_RESPONSE										X	X		X	R			
RECEIVE_PBX_SIGNAL													R	X	X		SWITCH
INSTALL_SWITCH_EQUIPMENT			R												X	X	
COLOR_TRUNKS															X	X	
COLOR_TERMINALS															X	X	
CONNECT_CALL														X	X	X	
DISCONNECT_CALL														X	X	X	
UPDATE_PERFORMANCE_MEAS						R											X
INSERT_INSTRUMENT																	X

CONNECTIVITY MATRIX 11/29/13

Figure 7-14. Resource, Process, and Module Connectivity Matrix.

However, in scenarios where each instance is heavily loaded, all of the processors will be working to simulate the inherent parallelism of the actual system. With this architecture, attempts at load balancing will likely slow the simulation. As shown below, there are typically many threads in an instance. Parallelization of these threads is handled automatically and not of concern to the designer.

The radio modeled in Figure 7-15 is typically one of many pieces of equipment on a platform. Platform models may be instantiated on the order of 500 to 1000 times. Each platform may have multiple instances of 5 to 10 different models that share information directly and are best placed on the same processor as the platform. As long as the number of processors exceeds the number of platforms, speed multipliers can be high. But designers need not be concerned with such decisions. They are automated in the Run-Time System as described below.

PARALLEL PROCESSING RUN-TIME CONSIDERATIONS

Instances of modules such as those shown in Figure 7-15 may be allocated to many parallel processors. To understand how this is accomplished at run-time, we summarize the following definitions:

- Two *processes are independent* if they share no resources.
- *Elementary Modules* contain only resources and processes.
- *Hierarchical Modules* contain Elementary Modules or lower level Hierarchical Modules.
- Two *modules are independent* if all of the processes in one are independent of those in the other.
- Processes may invoke other processes using the CALL or SCHEDULE statement, see [67].
- When a process CALLs another process, control is transferred immediately to the called process. Control returns to the calling process when the called process terminates.
- When a process SCHEDULEs another process, the scheduled process is identified in the schedule queue to be run when it is next in the ordered sequence of processes in the queue. The queue is ordered by priority within time as specified in the schedule statement, see [67].
- A *thread* is initiated when the next *scheduled process* is popped from the queue. A thread continues to run when a process in the thread *calls another process*. It terminates when the scheduled process terminates.
- A thread may span multiple modules within a single IND module.
- A module may contain multiple threads.
- Two *Threads Are Independent* if they are contained in *independent modules*.
- Threads contained in independent modules may *run concurrently* if the modules that contain them are allocated to separate processors.

Now consider the instanced model (module) in Figure 7-15 using the above definitions. As indicated, a simulation may have 300 instances of the mobile radio active in a scenario. By design, there may be on the order of 20 threads within an instance that are independent of those in the other instances.

If an instance is assigned to a single processor, none of the threads in that instance can run concurrently. Therefore there are no concerns about synchronization among these threads. Since threads in different instances are independent by design, there are no concerns about synchronization among those threads in different processors. The only case of concern is when a resource is shared across processors. In this case, the run-time system that gets generated maintains synchronization automatically, ensuring that processes sharing resources across processors cannot run concurrently.

In a heavy traffic scenario, most of the radios will be active at many of the protocol layers, implying that the activity on those processors will be relatively high. Additionally, the amount of time spent waiting for access to resources at the interface between modules will be small compared to useful processing time within an instance. As a result, the overlap of useful time should be high relative to the inherent parallelism in the system, implying that the speed multipliers should also be relatively high, see [40]. Without the visualization of architectural design that resides within this CAD approach, producing architectures that achieve comparable effective use of parallel processors is extremely difficult.

LOOKING AHEAD

The material covered above is intended to provide an overview of the following chapters. These next chapters will provide more detail on the underpinnings of a software theory that is aimed at taking maximum advantage of the potential speed to be gained from parallel processing. Having understood this theory, one should feel confident about new approaches to computer design, including basic processor design, chip design, OS design, and the design of a development environment that substantially aids both the hardware and software designer.

CHAPTER 8

AN INTEGRATED SOLUTION APPROACH

8.1 INTRODUCTION

The need for an integrated software development / run-time environment is motivated by requirements to significantly improve productivity and run-time speed on parallel processors. Trying to meet such objectives using current programming languages is a challenge that software developers no longer need to confront. This chapter describes an approach to building software that follows from engineering principles. Having used this system (known as VisiSoft), it becomes clear that design of the development environment must be integrated with that of the run-time environment. The approach described here achieves the following objectives.

- Substantially simplify software development for parallel processors.
- Create software that runs much faster on single as well as parallel processors.
- Control the growing complexity of a software system as it is expanded.
- Create modules that can be changed with minimal effects on the rest of the system.

SOFTWARE-HARDWARE ENVIRONMENT - FUNCTIONAL REQUIREMENTS

Figure 8-1 illustrates a decomposition of the software-hardware environment from application requirements to results. To address the objectives in the introduction, we must consider the individual requirements in the chain of elements in the software-hardware environment. These are described below, refer to [44].

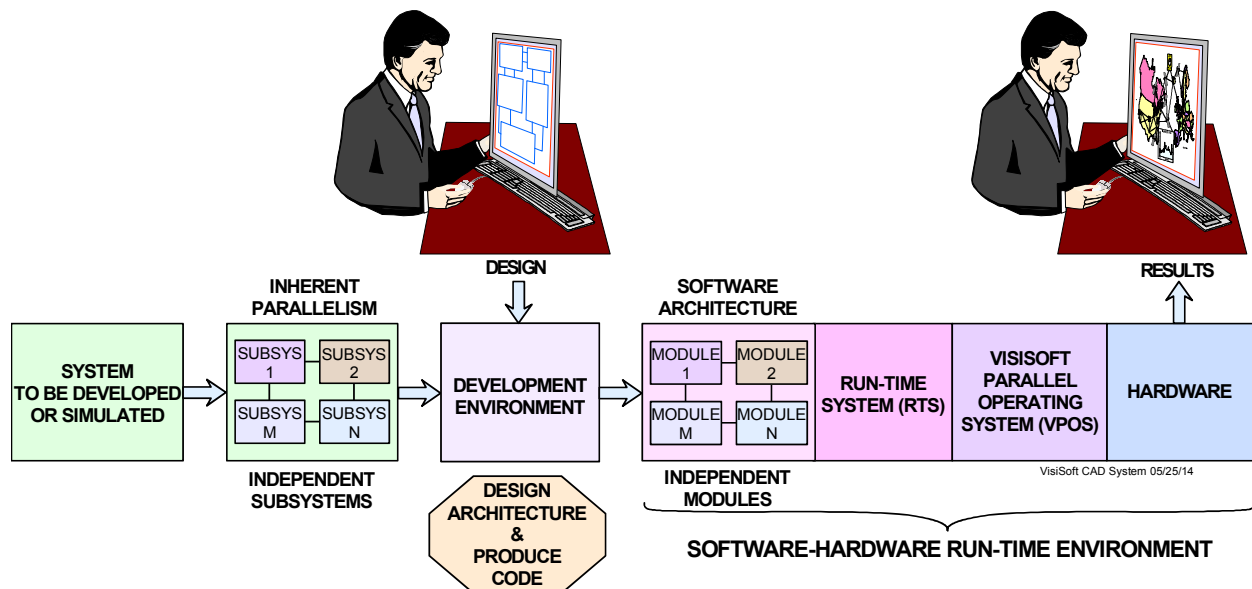


Figure 8-1. Overall software - hardware environment.

Application Requirements

This approach addresses large complex parallel processor applications requiring a team effort. For the purposes of this book, applications are divided into three types:

1. Embarrassingly Parallel - Applications that may be split into independent tasks that run concurrently with effectively no exchange of information.
2. Partially Independent - A single task that may be split into independent modules that must exchange information during the task, but where the processing time for information exchanges are small compared to what is going on inside the modules.
3. Effectively Sequential - A single task where most instructions follow from the prior ones, providing little chance for concurrent processing.

VisiSoft addresses partially independent applications, i.e., those with a reasonable amount of inherent parallelism. Being able to *represent a system's inherent parallelism in a software architecture* is key to the effective use of a parallel processor to meet stringent run-time speed requirements. Examples are real-time planning and control systems used in large manufacturing plants, or simulations of many platforms, e.g., aircraft, exchanging information - by radio - that affects their future behavior. The applications addressed also require high reliability, rapid enhancement to support new features, and potential for growth of complexity.

Architectural Design

For applications to be run on a parallel processor, architects must decompose the application into sets of relatively independent subsystems. These must then be translated into independent software modules such that processing within the independent modules far exceeds communications between modules. This is based on the inherent parallelism in the system. These independent modules may then be placed on separate processors to run efficiently.

For complex systems requiring special skills to produce the subsystem decomposition (e.g., systems requiring detailed engineering knowledge or special experience), subject area experts must be able to understand the software architectures with minimal, if any, help from programmers. They can then help design architectures that take full advantage of the inherent parallelism in the application system, something only they may have the knowledge to do.

Development Environment

The development environment must support high productivity to minimize the time and cost of development, validation, and testing. This implies rapid translation of application requirements into software architectures that reflect the inherent parallelism in the system. This is particularly true during post development upgrades and support.

This implies that architectures can be easily inspected, visually - using engineering drawings as in Figure 8-2. These show connectivity (they are not flow charts), to maintain full control over the design. It also implies that the languages (resource, process, and control specification) effectively support this architectural breakout. In addition, the languages must be easily read directly by subject area experts, so they can understand and validate complex algorithms representing the system as well as the architectural breakout and run-time control.

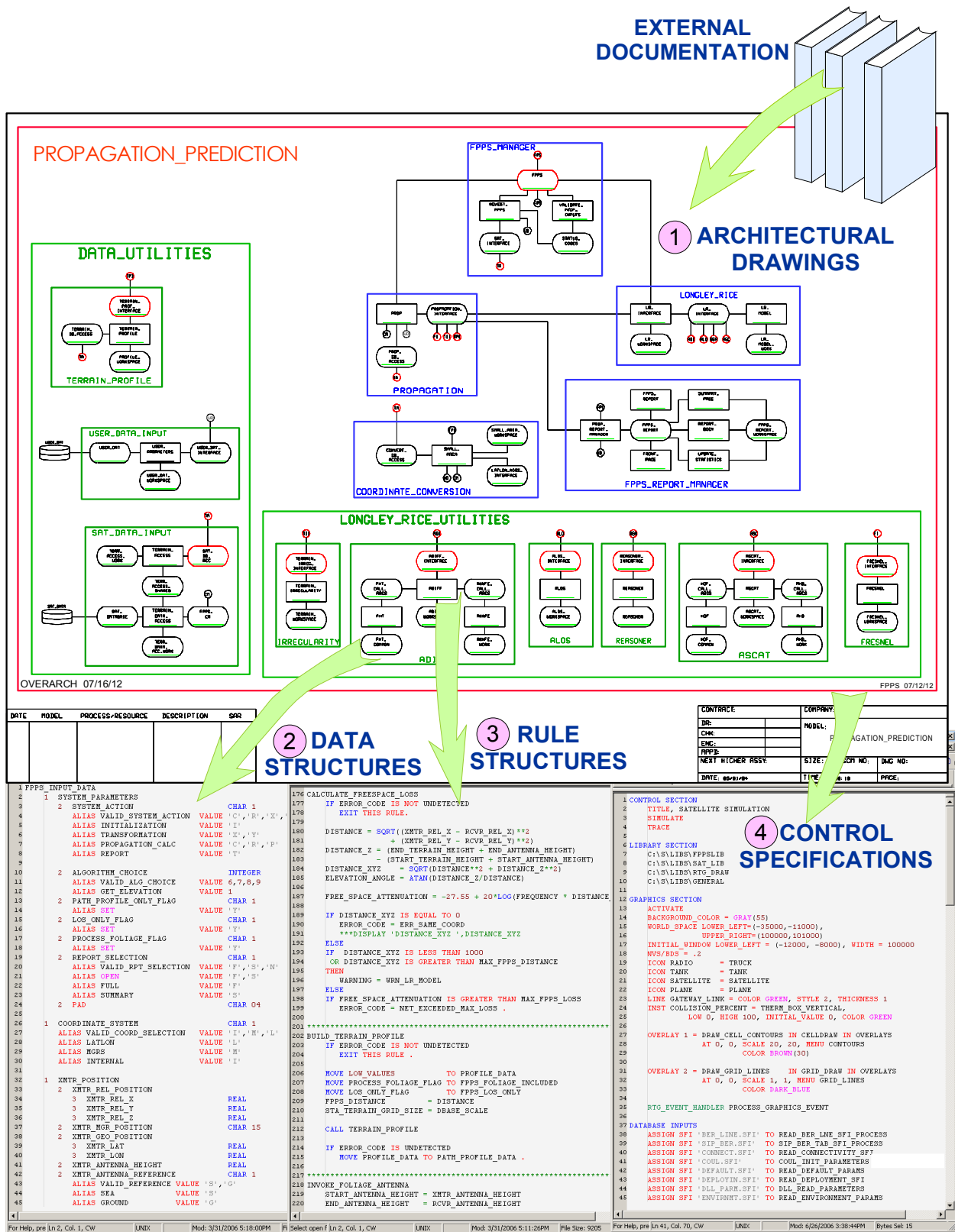


Figure 8-2. An overview of the development process using architectural drawings.

The development environment must produce the information needed by the run-time environment to ensure that full advantage is taken of the architectural characteristics of the application software. This is especially true when trying to achieve high run-time speeds on a parallel processor while minimizing the machine resources required to achieve that speed. This information is defined in the Control Specification, and defines the independent modules that may be assigned to separate processors, the number of processors, and a ΔT_{max} .

This environment produces the connectivity properties between modules so those that communicate may be located on physically adjacent processors to minimize communication delays. In the case that the use of these connectivity properties is nonstationary (modules may vary their connectivity properties by communicating with different modules as they operate), modules may be migrated during run time to reduce communication delays.

To support the above, the development environment must produce the application software object code in segments, corresponding to the independent modules produced by the architecture. Similarly, it must produce the database describing the independent module architecture along with management software to interface with the OS. Given this information, the OS can take maximum advantage of a (potentially simplified) hardware architecture.

Application Software

It is essential that the resulting application software be able to run fast on a single or parallel processor while using minimum machine resources. This implies that the machine code is organized such that hardware resource management, and in particular memory management, is simplified. This implies that the chunks of code to be managed are well defined and organized into a minimum number of chunks. This is another architectural design problem that depends heavily on the language used in the development environment to describe the databases.

Run-Time System (RTS)

The run-time system must provide the translation of architectural information from the development environment into calls to the OS during run time. It is the architectural design that minimizes the movement of instruction memory as well as data memory at run time. As indicated above, architectural information can be used to optimize processor allocation so as to minimize memory boundary crossing delays. Use of this information by the run-time system is critical to effective use of parallel processors.

VisiSoft Parallel Operating System (VPOS)

VPOS must be designed to take full advantage of the information provided by the run-time system. Specifically, it must be designed to allocate and assign hardware machine resources to make maximum effective use of this information. This includes minimizing overhead and memory sharing delays to achieve maximum run-time speed. This can only be achieved by allocating processors and memory to independent modules based upon the architectural information, including the possible migration of independent modules when the time-constants of nonstationary inter-module communications permit.

Hardware

In applications where run-time speeds are critical and parallel processors are required to support a single task, the hardware design must support the run-time system and corresponding OS requirements. In general, one typically trades memory for speed, duplicating instruction sets and stationary databases on separate processors to avoid swapping and paging. With the approach to architecture described here, hardware designers can focus on the essentials of minimizing overhead and memory sharing delays to achieve maximum speeds on a parallel processor, with little concern for the inherent architecture of an application software system. This is because full knowledge of the inherent parallelism of the system is embedded in the architectural design and automatically transferred to the run-time system.

With the integrated approach described here, the software development environment directly impacts the design of the run-time environment, including the OS. This, in turn, can be used to simplify design of multi-core chips. Specifically, the combination of language facilities and architecture eliminates the need for the hardware facilities in the bullets below, opening up chip real-estate for better use, e.g., more memory.

- Cache coherency
- Thread synchronization
- Stack facilities
- Special instruction swapping facilities

In the case where parallel processors may be dedicated to algorithm-intensive or memory-intensive applications that consume substantial processor time, they may be connected to server chips via shared memory as illustrated in Figure 3-6. When properly housed with a shared memory server environment, parallel processor chips need not interface directly with disks, communication channels, graphics, work stations, etc. One-way memory transfers to and from the server replace the need for special DMA channels or device interfaces.

Autos are a form of transportation vehicles, as are boats and airplanes. Each is designed for a different type of application. Similarly, servers are a different form of computer than are PCs or parallel processors, addressing different sets of applications. Our focus here is on parallel processors to support dramatic speed improvements when running large scale software systems and simulations.

8.2 ACHIEVING SPEED INCREASES

Design of the language for VisiSoft was driven by speed and accuracy for discrete event simulations of physical systems, typically with a high degree of inherent parallelism. The principle requirement was to develop a language that made it easy to build complex software for parallel processors as well as ease of understanding by subject area experts. The first step in the design was to separate data from instructions at the coding level. This *Separation Principle* simplifies the ability to track which sets of instructions share what data.

To minimize the number of data elements to be tracked requires the ability to support large hierarchical data structures. Similarly, one wants large hierarchical rule sets within a single process (a GSS process becomes a group of assembler instructions). Given that blocks of data are separated from blocks of instructions at the language level, one can easily build independent modules that map into the inherent parallelism of an application. As a by-product, this provides the ability to visualize the design using engineering drawings showing the connectivity of blocks of instructions with blocks of data.

Software Decomposition - Creating Independent Modules

The decomposition of a software system into independent modules implies drawing boundaries around the elements in a system that comprise a specified module. This allows any system to be decomposed into a set of modules. Furthermore, as modules get large, they can be decomposed hierarchically into submodules, etc.

Creating modules that can run concurrently on a parallel processor presents explicit requirements on module design. Two modules can run concurrently only if they are *independent*. This implies that they share no data; else they incur the potential for inconsistent use of that data. The independence property is also an important contribution to the other requirements stated above.

Multipliers On The Speed Multipliers

Being able to easily define and reference large data structures allows them to be moved using a single instruction fetch into another shared structure that defines the details of all of the elements. This provides for significant increases in speed when working with algorithms requiring large state vectors or databases. This has been demonstrated in a substantial number of case histories and experiments.

Given the speed multipliers that VisiSoft has generated on single processors, one may expect to use fewer processors (as many as a factor of 10 less) simply by using the VisiSoft environment to build the software.

Using the architectural features of VisiSoft, one may create larger independent modules that will run faster (using less overhead) provided that each processor has sufficient adjacent memory. This new architectural approach affords speed increases that require fewer processors to achieve the same speed multiplier.

Using fewer processors reduces the distance between processors, further increasing the speed multiplier. This is clearly a nonlinear function, where speed increases with fewer processors. Conversely, speed will decrease nonlinearly with more processors if they increase the overhead. This has been shown to be true in many parallel processor experiments.

Elimination of Data Coherency Checks

When one independent module wants to communicate with another, it must use an Inter-Processor (IP) resource. The Inter-Processor Communication (IPC) system contained in the VisiSoft Run-Time System (RTS) automatically copies an IP resource into a similar IP system data structure, or from an IP system data structure to an application IP resource, to protect a copy of the data. Architectural rules built into the system prevent developers from having two processes write into the same IP resource. This, coupled with copies of memory, eliminates user concerns with data coherency. Need for time-consuming data coherency checks at the system level are eliminated by virtue of using memory copies that also improve speed, and architectural rules that simplify the architectural design.

Scheduling Of Threads

VisiSoft Threads are used to define a hierarchy of processes within a task. All processes run as part of a thread. Threads contain one or more processes. A thread is SCHEDULED to run when its lead process is scheduled to run at a given time based on a simulation or real-time clock. When the clock advances, and the lead process is the next to run in the schedule, that process (and corresponding thread) is started by the scheduler. The lead process may CALL other processes that in turn may CALL others. Called processes are contained in the thread.

On a single processor, no other threads within a task may run until the current thread completes. While a thread is running, it may schedule other threads to run at a later time, or NOW. When scheduled NOW, the thread runs at the same clock time as the current thread, after it completes. Threads run in a sequence defined by their scheduled times and a priority code when used. When both the time and priority are identical, the outcome is considered random.

In a parallel processor environment, threads in the same task may run concurrently on different processors. VisiSoft provides facilities to simplify synchronization of threads running in parallel. One of these facilities is the Independent (IND) Module - further described below. IND Modules are contained within a single processor, and a thread must be contained within a single IND Module. Therefore, threads within an IND module cannot run concurrently since they are on a single processor.

Threads in one IND module may schedule threads in another (or the same) IND module. All threads are controlled by the local Scheduler and the Synchronizer which ensures those on separate processors do not get out of synchronization. Thus, the developer has no concern for synchronization of threads, delays or race conditions. Special run-time facilities exist that allow timing to be out-of-sync up to a ΔT_{max} , where ΔT_{max} is determined based upon comparing error distributions of simulated results with live test data or single processor systems or simulations. This is further described below.

The VisiSoft CAD environment may be used to develop any type of software system. It is specifically suited to developing the most complex types of systems. It is well suited to development of an operating system, with its built-in schedulers, synchronizers, sort facilities, and 3D graphical facilities in many coordinate systems. With sophisticated libraries for handling hierarchical link lists, it is well suited for building database management systems and applications supported by servers.

8.3 APPLICATION CHARACTERISTICS

Applications amenable to parallel processing fall into various categories. We must characterize applications in terms of their:

Inherent Parallelism - The inherent properties of an application that determines the potential processor utilization efficiency

Software Architecture Potential - The ability to translate inherent parallelism into independent modules to achieve maximum processor utilization efficiency.

Categorizations of interest are listed below. They are described in terms of factors that affect relative processor utilization efficiency for modules that run concurrently.

- Module Size - The size of a module relative to the time spent running concurrently.
 - Fine Grain - Fine Grain modules may require a high degree of overhead (relative to concurrent processing) for communications and control.
 - Medium Grain - Medium Grain modules may require a fair degree of overhead.
 - Large Grain - Large Grain modules may require relatively little overhead.
- Module Connectivity
 - One-To-Few / One-To-Many / Many-To-Many / All-To-All - This indicates the degree of information exchange requirements between modules, implying the degree of memory sharing required.
 - Stationary / Non-Stationary - This indicates whether the connectivity is changing during the course of a scenario. The time-constants of change are significant factors in the Non-Stationary case.
- Module Interaction
 - Linear / Quasi-Linear / Nonlinear / Highly Nonlinear - This implies how modules affect each other's behavior. In the linear case, the effect can be embedded in each module using superposition. In the quasi-linear case, one can use transformations to obtain a linear representation. In the nonlinear case, one may have to iterate between modules to determine the matching solution. In the highly nonlinear case, iterations become more prevalent.
- Module Scenario
 - Lightly Loaded / Fully Loaded - This determines the level of module activity. Modules may do very little in a lightly loaded scenario, cutting processor utilization efficiency while achieving minimum run-time. In a fully loaded scenario, processor utilization efficiency may increase so that run-time may be diminished by a relatively small amount.
 - Stationary / Non-Stationary - This will determine the time constants of change.
 - Fixed / Variable - Some trails of a scenario may follow a relatively fixed path while others take totally different paths with significant differences in run-time.

Inherent Parallelism - The Extremes

Looking at the extremes, if there is no inherent parallelism in a system, then each instruction in the corresponding simulation or software task depends upon the prior one, and one is left with a single module. A parallel processor will only slow things down. If the task is embarrassingly parallel, there is virtually no connectivity between the modules (no memory sharing) and one can use a cluster or a server environment to process independent tasks. However with substantial parallelism but also communications between substantially parallel parts, one must use a Single OS (SOS) parallel processor.

Reading And Writing Large Files

In most simulations - with few exceptions, parallel processors are required to cut single processor run-times by orders of magnitude. The parallel processors needed to run simulations typically do not need to interface with DMA devices directly, but instead initialize large databases in a Read Only Mode (ROM), and produce output in a Write Only Mode (WOM). Additionally, filling large databases is typically done before the simulation starts - during an initialization period. Dumping memory for output usually occurs in small chunks. These I/O requirements are defined by the architecture of a system and can be handled by sharing memory with a server - to fill memory coming from a device and to dump memory being sent to a device. Direct interfaces with the devices are unnecessary, being an insignificant part of the processing time. This form of memory can be shared between the parallel processor and the server. This eliminates the need for device drivers and DMA channels in the parallel processor, providing more chip space and memory for direct processing.

Some Important Concepts Regarding OS And Chip Design

Prediction accuracy is determined by conditional probabilities. The more information one has to condition the probability statement, the more accurate the predictions, see [40]. This applies directly to the parallel processor software design problem, both for the development environment and the run-time environment.

For example, the ability to create software architectures that represent the inherent parallelism in a system is critical. Similarly, the information that the run-time system can use about that software architecture can dramatically improve processor utilization efficiency. This depends upon the environment used to develop that software as well as the architect who must translate the inherent parallelism of a system into an architecture that properly represents it using Independent modules that are recognized by the OS at run time.

In the simulation applications of interest here, processors are allocated as a group to the simulation task. These processors are not multi-tasked. Only the OS on those processors may run and share their resources.

There must be a master OS on the parallel processor that controls the allocation of all processors and the memory they use and share. Local subsets of that OS must reside on each processor controlling the use of that processor and its memory. A local OS may copy memory from another processor by going through the memory manager.

If all of the code and data of an independent module reside in local instruction memory and data memory (level 1 cache), swapping and paging are unnecessary. This implies that (1) they fit; and (2) they are not going to move (stationary connectivity). Alternatively, if they do not fit, but the statistics are still highly stationary, time spent swapping and paging will be insignificant. This affects significant trade-offs between memory size and special hardware algorithms for swapping and paging that use chip space. With enough memory in each of the areas in the memory hierarchy, swapping and paging time will be insignificant.

By creating independent modules during the architectural design, where threads are always independent between modules, and are sequential and cannot run concurrently within a module, thread synchronization is unnecessary. Using hardware (chip space) and OS code (memory) for this function is unnecessary.

Similarly, since use of IP resources on separate processors is synchronized automatically by the run-time system, there is no need for coherency checks. Again, using hardware (chip space) and OS code (memory) for this function is unnecessary.

Finally, it is not clear how the use of a stack saves time, unless it is used to track recursive use of processes. Since this is not allowed (and certainly not needed in any software or simulation that PSI has built over the last 50 years), it appears to be unnecessary for the applications of interest here, likely to slow things down, and wasteful of chip space.

Hardware Effects

From the above, one sees that the software development environment affects the design of the run-time environment, including the operating system. Together, they both affect the design of the hardware. Specifically, the following hardware facilities may be eliminated from the parallel processor chip for the applications of interest and using the approach proposed here.

- DMA channel/device interfaces
- Cache coherency
- Thread synchronization
- Stack facilities
- Special instruction swapping facilities

Potential Speed Multipliers & Processor Reduction

One must consider all of the factors that affect speed when comparing the VisiSoft CAD approach to current “advanced” approaches. For example, we must consider applications with a reasonable degree of inherent parallelism (greater than 50%). We are also concerned with heavily loaded scenarios where single processor time is generally the longest. When this occurs using VisiSoft, the percentage of idle processors is typically relatively smaller, making processor utilization efficiency higher. Then a large number of processors may be unnecessary.

Figure 8-3 shows the Maximum Speed Multipliers one can obtain from a parallel processor based on the Inherent Parallelism in a system and the number of processors applied to the task. Clearly the Inherent Parallelism plays a major role in determining the multipliers. But this chart is easily deceiving since it is based upon the *maximum* one can achieve. The true outcomes will depend upon the software architecture and the environment that one has to develop a good architecture. Given that the inherent parallelism in a system is in the 70-90% range, one must translate that inherent parallelism into a multiplier for a given number of processors.

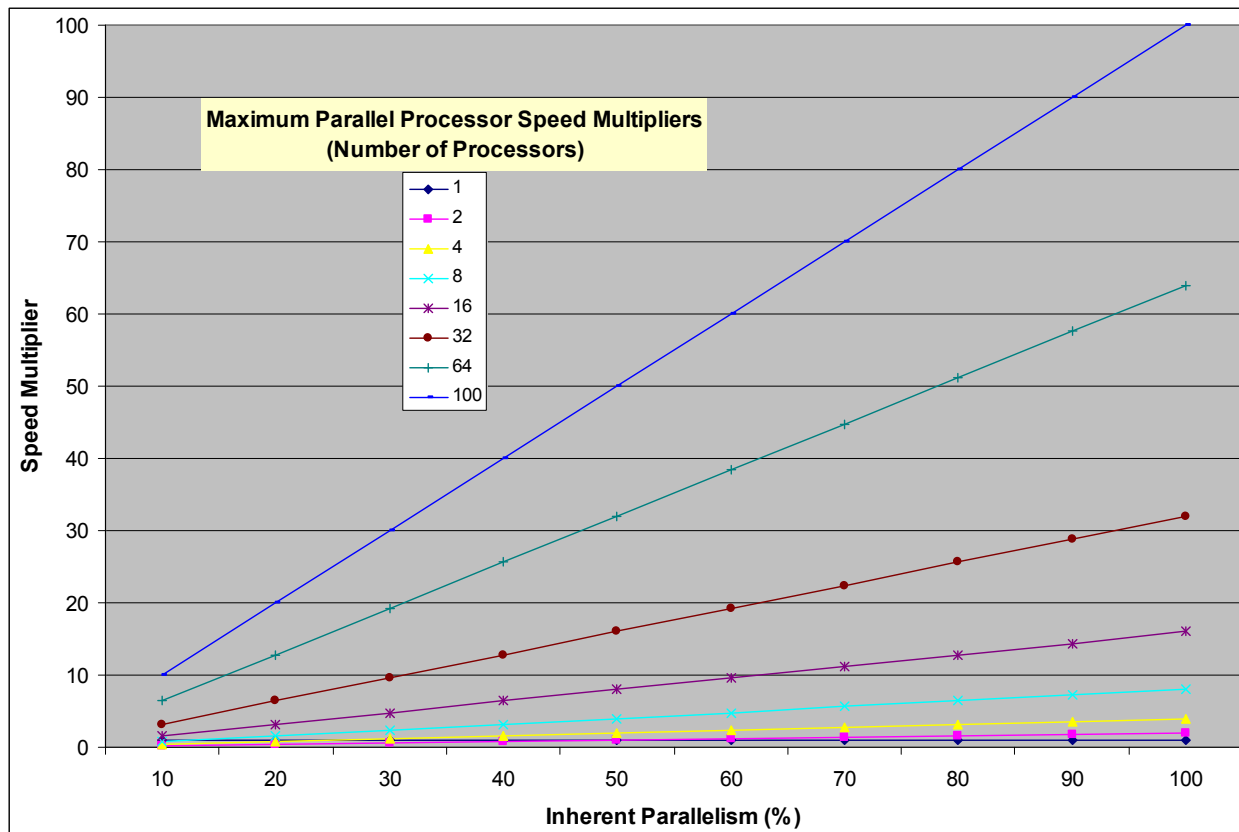


Figure 8-3. Plot of Maximum Speed Multipliers vs Inherent Parallelism and Processor Count.

Very little comparative test data is shown in the recent literature on this subject. What is available indicates that current experience has not changed much from prior published results and may have gotten worse. Current interest appears to be focused upon the number of processors that can be put together as a single computer, followed by claims of a high *potential* speed multiplier. But actual test data has shown that the efficiencies typically go down nonlinearly with the number of processors. Much of this is due to relatively poor software development environments, and the resulting spatial footprint of the computer. As the software design is improved, more processors can be put into a smaller spatial area, further increasing speed. To see this, the actual speed multiplier is the important measure that must be compared.

From a user market standpoint, there are two aspects of speed improvement that one can achieve from VisiSoft. First is the case where one simply increases the speed of a given software system or simulation. In this case, a simulation may run for 10 minutes instead of 2 hours. In the other case, the speed requirement is fixed but the requirement is to minimize the number of processors. In the latter case, a nonlinear reduction occurs for any given processor design since the spatial footprint may be reduced considerably, reducing transmission delays as well as memory boundary crossing delays. We will use the reduction of number of processors to understand the factors affecting the potential speed improvements from VisiSoft. This is a critical point that takes advantage of multiple factors and provides potentially large returns, including significant reductions in power consumption and floor space as well as hardware costs. The major factors of concern are the following.

- Single Processor Speed Multiplier - This is the difference in speed using VisiSoft versus current software development environments on a single processor.
- Parallel Processor Software Architecture - Using VisiSoft, one can produce a map of the inherent parallelism of an application system into an optimized architecture of IND modules. VisiSoft IND module architectures directly affect large increases in Processor Utilization Efficiency (PUE) with more useful work done on each processor.
- IND Module Mapping - VisiSoft IND modules are generally large and remain on a specified processor, typically eliminating swapping and paging. If the number of IND modules is larger than the number of processors, multiple IND modules may be put on a single processor. VisiSoft also provides run time measures of PUE for each IND module for each processor. As described in Chapter 18, this information can be used effectively for grouping multiple modules with relatively small utilizations onto a single processor. It can also be used to breakup modules that take a lot of time, and can be run in parallel on separate processors using less time in the ΔT_{max} window. This reduces the overall idle time within the overall ΔT_{max} window. Both of these assignment approaches can significantly increase the PUE.
- VisiSoft Parallel OS Speed - This is the difference between a Windows or Linux OS and VPOS. Depending on the application, VPOS runs from 2 to 10 times faster than Linux or Windows and takes full advantage of the VisiSoft Run-Time System (RTS).
- Better Use Of Chip Space - VisiSoft maps IND modules into separate processors and eliminates designer concerns for thread synchronization. Communication between processors uses the run-time IP Communications (IPC) manager eliminating concerns for synchronization. Sharing memory with a server eliminates the need for a DMA channel interface to external devices. Stack facilities and complex instruction caching are eliminated. Cache coherency is of no concern when using VisiSoft. All of these serve to simplify the chip design allowing for more memory close to the processors, further reducing swapping and paging.
- Speed-Distance Factor - This is the result of the reduced distance between processors and memory due to the above factors producing the same speed multiplier with a reduced number of processors.

When minimizing the number of processors, the final speed multiplier depends upon the above factors. Estimates of the low, expected and high values for three factors derived from the above are provided in Table 8-1. These are: Single Processor Speed Multiplier; Processor Utilization Efficiency; and Distance Factor. As indicated above, these values depend upon various other factors, e.g., the number of processors, and the size and intensity of the scenarios.

Table 8-1. Factors affecting speed multipliers and processor reduction.

COMPARATIVE SPEED MULTIPLIERS†											
Single Processor Multiplier			Processor Utilization Efficiency			Distance Factor			Final Speed Multiplier†		
Low	Expected	High	Low	Expected	High	Low	Expected	High	Low	Expected	High
2	4	6	2	3	4	2	3	4	8	36	96

† As shown in Figure 8-4, these Speed Multipliers are approached as the number of processors gets large.

The factors in the table must be derived from experiments using applications of interest. Based upon prior experiments, the values in the table are considered to be conservative estimates for problems that are not embarrassingly parallel. The multipliers in Table 8-1 do not include the efficiencies provided by VPOS, nor improved chip designs that may be obtained from the VisiSoft approach. The details used to derive Table 8-1 are explained in the sections below.

Single Processor Multiplier

Based on many comparisons, the typical single processor speed multiplier range is from 2 to much more than a factor of 10. Chapter 17 has experiments that show how a high value of 78 is obtained when comparing VisiSoft to C-based languages. The expected multiplier of 4 is small compared to prior comparisons of VisiSoft to FORTRAN or C-based languages.

Processor Utilization Efficiency

In the table above, different values of PUE for a competitive approach are used to obtain a ratio for the Final Speed Multiplier. For example, the low multiplier of 2 is obtained using a VisiSoft PUE of 0.4, and the competing PUE used is 0.2. For the expected multiplier of 3, a VisiSoft PUE of 0.6 is used over the competing PUE 0.2. For the expected multiplier of 4, a VisiSoft PUE of 0.8 is used over the competing PUE of 0.2.

These multipliers are derived from PUEs obtained when using VisiSoft versus typical results obtained from current approaches. High PUEs are obtained from VisiSoft IND module architectures and the automatic characterization of processor utilization that the system produces at run time. The improvement multipliers are simply expected ratios of VisiSoft PUEs to those from other approaches. A PUE of 80% is commonly achieved when using VisiSoft.

Distance Factor

If the number of processors is cut by a factor of 8, one may observe an additional speed multiplier of 2 or more, just due to the reduced distance between different processors and their memory. It is well known that delays due to distance increase nonlinearly.

Final Speed Multiplier

The final speed multipliers will depend upon the original number of processors being used in an existing application. VisiSoft provides run time measures of PUE for each IND module for each processor. Chapter 18 illustrates how this information can be used effectively for grouping multiple modules with relatively small utilizations onto a single processor. It can also be used to breakup modules that take a lot of time, and can then run in parallel on separate processors. This reduces the time it takes to process large IND modules in a ΔT_{max} window reducing the overall idle time on other processors within that ΔT_{max} window. Both of these assignment approaches increase the PUE.

Figure 8-4 provides an illustration of processor reduction using what are considered realistic representative cases derived from Table 8-1. We note that the reduction factors will depend upon a number of criteria. For a large number of processors, the reduction factor for numbers of processors can be expected to be as high as 50 and likely higher depending upon the difference in architectures. The chart shows that 1200 processors are reduced to 40. This corresponds to a reduction factor of 30 which is considered conservative compared to actual test results. This chart represents significant reductions in floor space, environmental equipment, and power consumption as well as hardware costs. Reduction in delay times are nonlinear because of the factors described above.

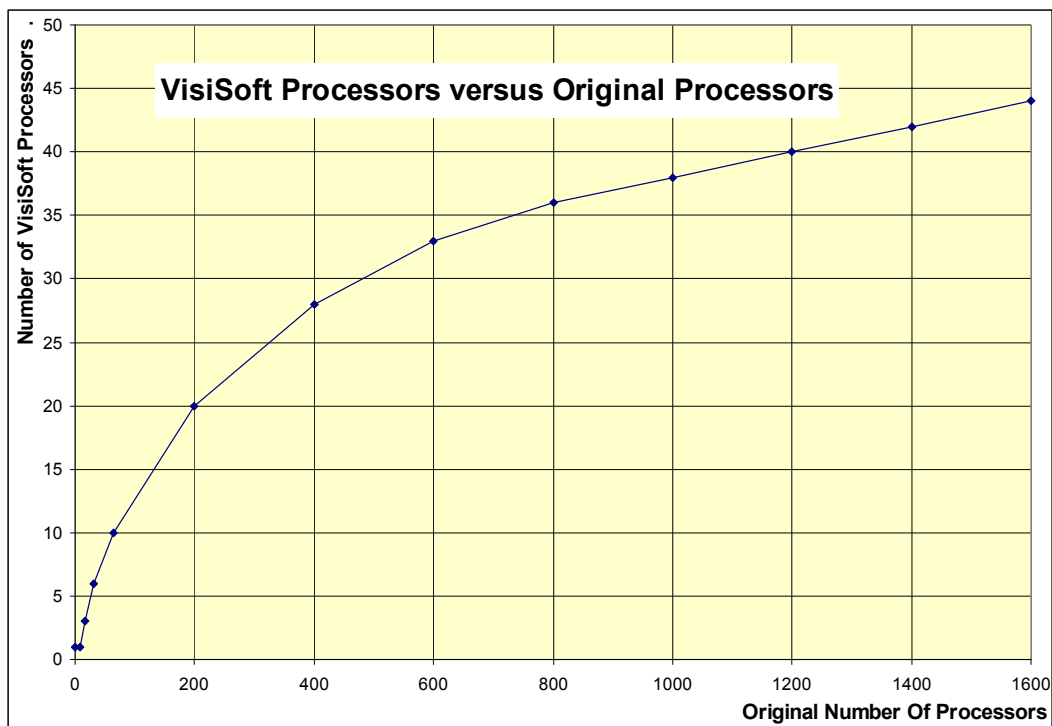


Figure 8-4. VisiSoft Reduced Number of Processors versus Original Number of Processors.

Note that the multipliers used to derive the curve in Figure 8-4 do not include the additional factors of efficiencies provided by VPOS, nor improved chip designs that may be obtained from the VisiSoft approach.

CHAPTER 9

SOFTWARE ARCHITECTURE FOR PARALLEL PROCESSING

Most readers will relate to the drawing in Figure 9-1. As in other fields, architecture is much more graphical than algebraic or textual. Whether designing machines, ships, or buildings, architects produce drawings. These drawings are neither “approximate” nor “abstractions”. They are precise engineering specifications that are followed into production.

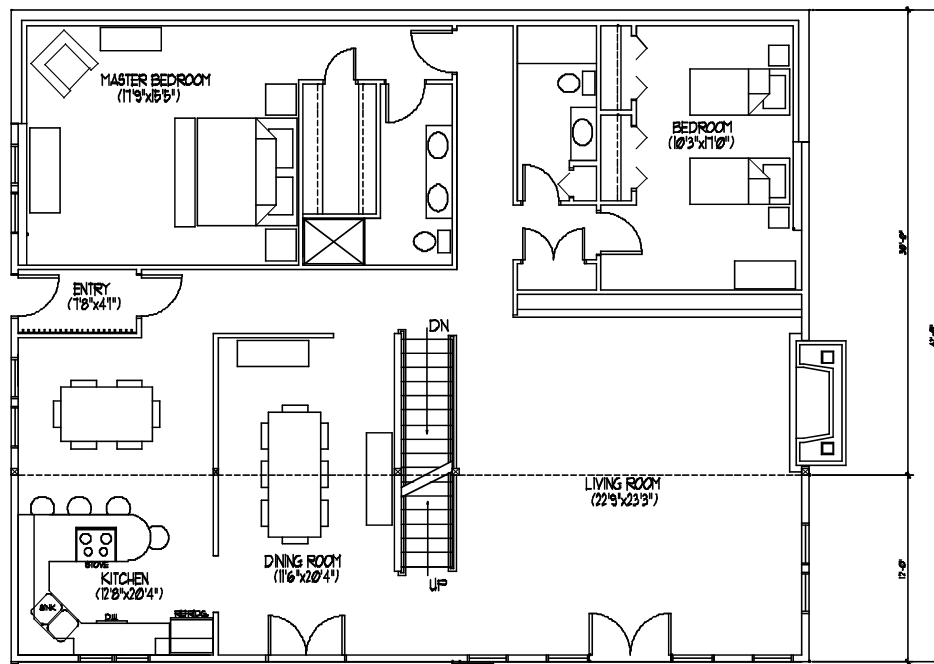


Figure 9-1. An example of engineering architecture.

Origins Of The Field Of Architectural Engineering

Detailed architectural plans required for modern complex building structures have their origins in the Renaissance period. In those days, artists made sketches of buildings that represented the plans they imagined. Their renderings did not include measurements, and the conventions required to support detailed plans for today’s complex structures had not yet been thought about. Builders were expected to follow the illustrations and work out the details.

As buildings became more complex, the art of drafting the plans was forced to advance to the point where everything was explained in detail. By working up the plans and including many levels of detail, the process was forced to change. Engineers were expected to solve all of the engineering and construction problems - before the actual building began. Creating detailed designs and plans avoided gross mistakes, misunderstandings, construction delays and corresponding cost overruns. It also helped the builder to develop accurate plans and cost estimates.

Improved tools such as adjustable squares and technical pens reduced the time and labor needed to produce the drawings. Technical drafting aides such as the parallel motion drafting table and transfer lettering also helped to reduce the effort in producing the drawings.

The greatest advance in creating architectural drawings came with the application of computer technology to this discipline. The production of building plans has been taken over by CAD software systems which have increased both the capabilities and speed of completion for planning structures. The choices for rendering details, selecting materials, and solving engineering challenges have been greatly simplified. Digital plotters have made reproducing complex drawings an easy matter.

To facilitate graphical representations of architecture, CAD tools are used extensively. The time to produce and reproduce drawings has been cut dramatically since the days of drawing each line by hand. Reuse of drawing parts is common - they are copied and modified easily.

The need for architectural drawings in software is motivated by most of same requirements as those in other engineering fields, especially increased productivity, leading to a high quality product. Important requirements are the ability to:

- Decompose a software system into modules that can be worked on independently;
- Create modules that can be changed with minimal effects on other parts of the system;
- Create modules that can be run concurrently on a parallel processor.
- Control the growing complexity of a software system as it is enhanced;

Engineering CAD facilities provide graphical interfaces that are designed to make it fast and easy to:

- Decompose system requirements into architectures;
- Recognize connectivity and corresponding independence of modules;
- Distinguish between good architectures and bad architectures;
- Plan for change.

It is hard to imagine an architect designing a skyscraper without engineering drawings.

ARCHITECTURE - A NEW SOFTWARE CONCEPT

The CAD tools referenced in the prior chapters provide a precise visualization of the architecture of a software system. This engineering approach provides a one-to-one mapping from the top level architecture to the code. Using the graphical CAD front-end, one can drill down - from the top system level drawing - to the details of the code, with no abstractions in between. The interconnection lines are as meaningful at all levels of a drawing as they are in electronic circuit design, logical design, or machine design, i.e., *no abstractions*.

This CAD environment has been derived from the same concepts used by chip manufacturers for designing hardware. It provides for decomposition of the architecture, and composition of the detailed design using graphical symbols that directly represent the software. With this approach, it is apparent that *architecture is the most important part of software design*.

Having used this CAD system, one cannot imagine working without drawings. One also observes that software architecture is only accomplished using the totally new paradigms described here. In software design courses using this approach, architecture is taught first - before language or coding facilities are described. With this approach it becomes clear that architecture has the most effect on productivity, especially in the support phase of a product. *Architecture is essential when designing software to run on parallel processors.*

SOFTWARE ARCHITECTURAL COMPONENTS

The basic architectural components of a software system have been introduced in Chapters 5 and 7. The use of these basic elements will now be expanded to create hierarchical modules as shown in Figure 9-2. These include resources, processes, elementary modules, hierarchical modules, utility modules, and library modules. Their architectural properties are described below. Toward the end of this chapter we will introduce the INDependent (IND) Module for parallel processing.

Elementary Modules

The module hierarchy in Figure 9-2 is apparent, down to the elementary modules shown as Drawing Layer 1 (Drawing Level 1). Elementary (Layer 1) modules contain resources (ovals - representing data structures) and processes (rectangles - representing rule structures). Architectural connections are designed at the elementary level to maximize independence between hierarchical modules. This allows reuse of modules in other hierarchies.

Hierarchical Modules

Figure 9-2 contains an illustration of a layered module hierarchy. At the bottom of the hierarchy are elementary modules, Layer 1, shown in Drawing Level 1. The module layers are determined as part of the architectural design. Drawing Levels are selected as a convenience for visualization of the architecture. Hierarchical modules are illustrated in Drawing Levels 2 and 3. As shown in the figure, a hierarchy of Module Layers may be contained in a single Drawing Level (Drawing level 1 contains 3 layers of hierarchy). It is not unusual for complex systems to take up to nine or ten layers of hierarchy. At this level of complexity, one may typically expect a system to contain millions of lines of code.

System Decomposition And Module Composition

The decomposition of a system into a hierarchy of modules requires an understanding of the particular application being developed as well as experience in software architecture. We note that, from a developer's standpoint, complex applications include the development of language translators and operating systems. Regardless of the application, the grouping of resources and processes into elementary modules is an important architectural design function. All resources and processes must lie within an elementary module boundary. This implies design of the module and its components including the interface resources between modules. Those responsible for that module have implicit control over that interface.

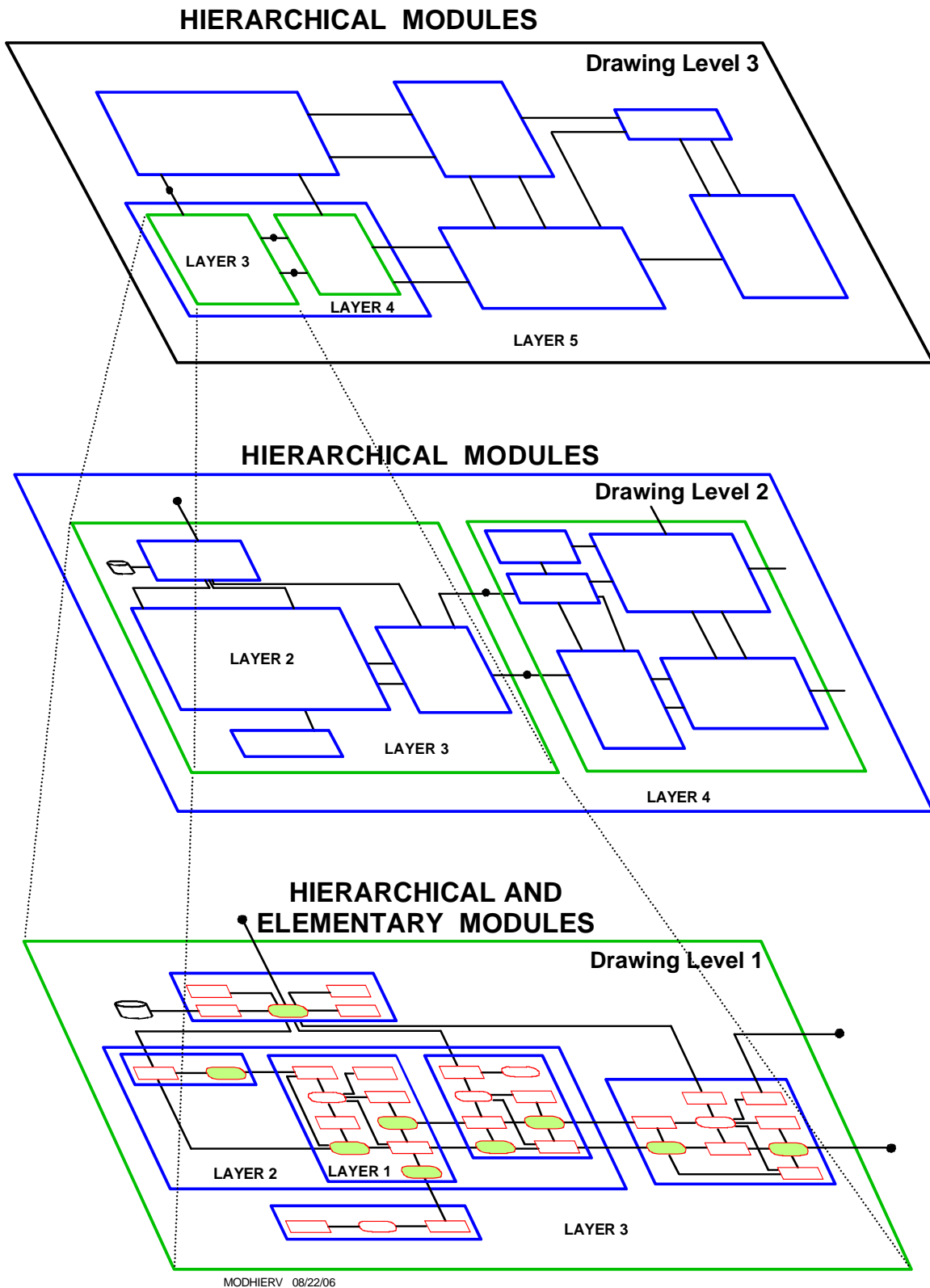


Figure 9-2. Illustration of a module hierarchy.

Most importantly, architectures need not be poured into concrete. On the contrary, using the CAD environment described here they are easy to change. This is because the processes, resources, and connection lines are moved easily from one module to another. So if one decides to move an interface resource from one module to another, it is a simple and very visible drawing change that confers control of the interface to the other module.

Module Types

There are three types of modules that may exist in any layer of a hierarchy. These types provide different levels of protection with regard to their reuse in different hierarchies. Both elementary and hierarchical modules can reside within each type. With the exception of instanced utilities, modules may only appear once in a drawing. The rules for these types are described below with examples in Figure 9-3.

- **Modules** - have a blue border. These are the basic building blocks in a task. In the CAD system described here, modules may be decomposed hierarchically, i.e., they may contain submodules and sub-submodules, etc. Modules may only appear in a single drawing in a user directory, and are meant to be unique, i.e., not reused, across directories.
- **Utility Modules** - have a green border. These are modules that are reused by processes in the same directory, and can appear in more than one hierarchy in different drawings. They are typically used to manage separate databases or perform utility type functions. The green color distinguishes them for change protection. If they are changed to accommodate a different requirement, that change must be compatible with the other processes that use them, since the change is automatically embodied in them all.
- **Library Modules** - have a gold border. These are more highly protected utility modules that can be shared from different directories and different computers. They are stored as object modules in special object library files. The source only appears in the directory where they are maintained. Processes in a library module are called from an application using their process name, module name, and library name. Since each of these names must be unique within the next level of hierarchy, there can be no duplicate names when linking to library modules in the CAD environment described here.

The functions of a library module may be upgraded while at the same time preserving the original module in the library for prior users. Users can call the new function using the same process name within the same library by using the new module name. The existing CAD system has a large set of libraries that support various applications, including 3D graphics, that are shared easily.

The CAD libraries have been designed to be controlled separately under special protection mechanisms. But given access to a library directory, the responsible person sees everything that is needed to allow for ease of changes and testing. Library directories typically contain regression test drivers and data sets to ensure changes meet all prior, as well as new requirements.

The top level module in Figure 9-3 is a library module. It contains modules and utility modules that are immediately identified by the color of their border. This library module performs electromagnetic wave propagation loss predictions between antennas using detailed terrain, foliage, and building data. Being a key module for exchanging data in a communication system, this architecture has some special properties that are used in the simulation described in Chapter 18.

USE OF ENGINEERING DRAWINGS

Although written documentation is important, in practice, engineering drawings are the essential tools to support the planning, review, and assessment process needed to control a large complex project. This is because of the requirement to develop and modify the structure of modules as the design unfolds. Figure 9-4 illustrates the ease with which one may drill down directly to the code of a complex module, knowing exactly where that module fits in the hierarchy, and what resources are shared with other modules. This is possible for any type of module at any drawing level.

Engineering drawings provide the means for creating and improving the structure of software. This is because, given the CAD tools described here, these drawings are modified easily to implement structural improvements. Also, the best structure for a complex set of modules cannot be known until most of the design has been completed and carefully reviewed with the software team. Only after understanding all of the facilities that must be built into a module, and how those facilities interact with other modules, can the developers decide on the best architecture for a module. This implies that a module may be built initially using an inadequate structure before one can see how to improve that structure.

In general, libraries may contain a virtually unlimited number of modules. Also, individual library modules can be huge hierarchies. Libraries are easily expanded by taking an existing module and adding new functions or modified versions of existing functions. When functionality can be reused, the use of library modules greatly simplifies software development.

SOFTWARE ARCHITECTURE

An overview of the architectural facilities incorporated into the CAD system for building and supporting multiple tasks on multiple computers is provided below. We note conversely that a single task may require parallel processing where modules running concurrently on different processors are not totally independent. Just as with other architectural facilities, one must account for all of the facets of the problem. Software architectures must support the user environment as well as the development and support environments. Although our immediate focus is on development and support, the run-time environment will be addressed in follow-on chapters.

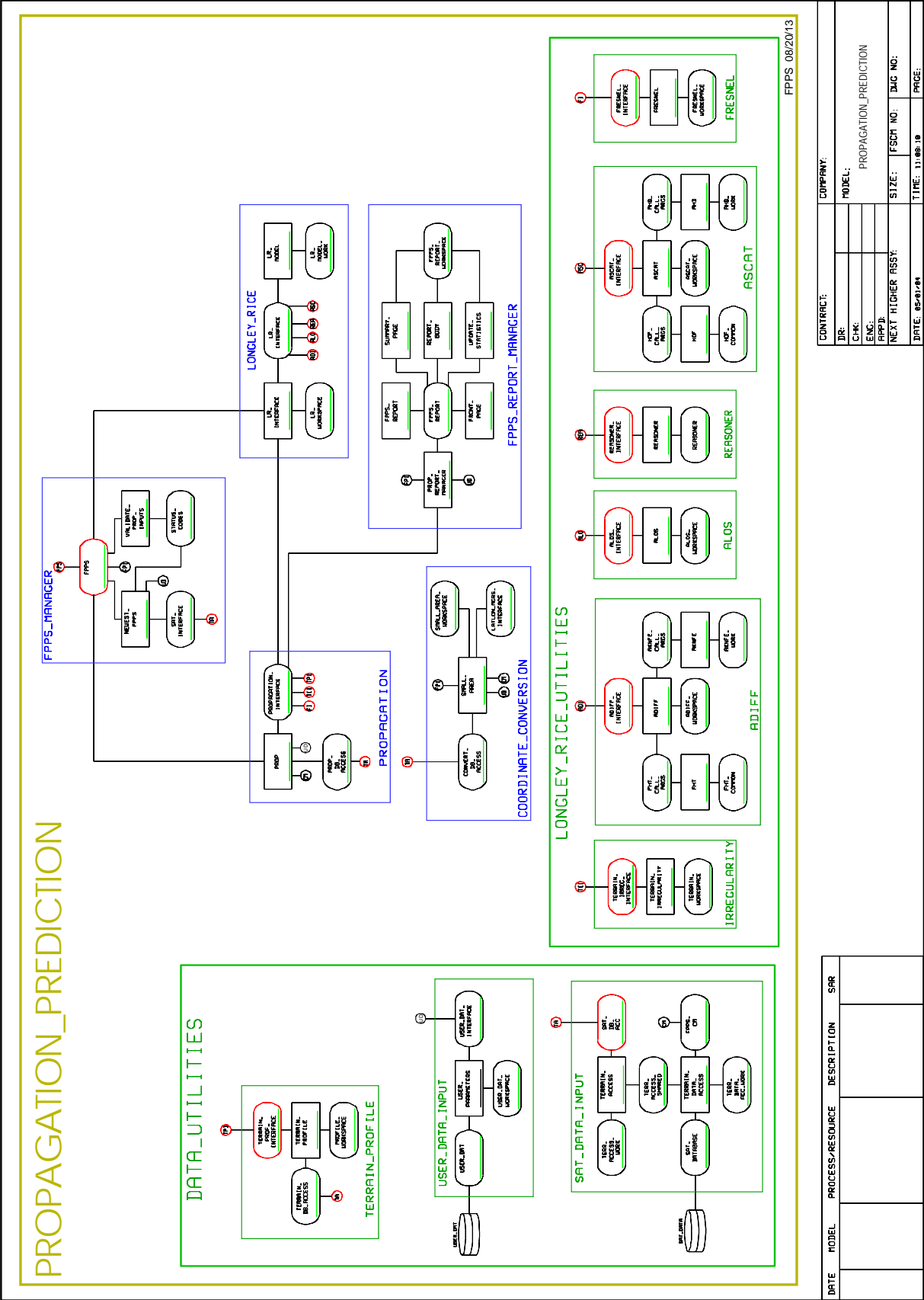


Figure 9-3. Engineering drawing of a library module.

Task Hierarchies

Tasks are executable modules at run-time. The logical system hierarchy illustrated in Figure 9-5 contains 5 tasks. Architecturally, a task can start one or more additional tasks, and a task can invoke one or more modules by starting a process. Modules within a task drawing may be elementary or hierarchical. There is no theoretical limit on the hierarchy. However, a task containing 8 or more levels of hierarchy is a huge piece of software (on the order of 1M lines of code). Modules in a task need not reside inside the task drawing.

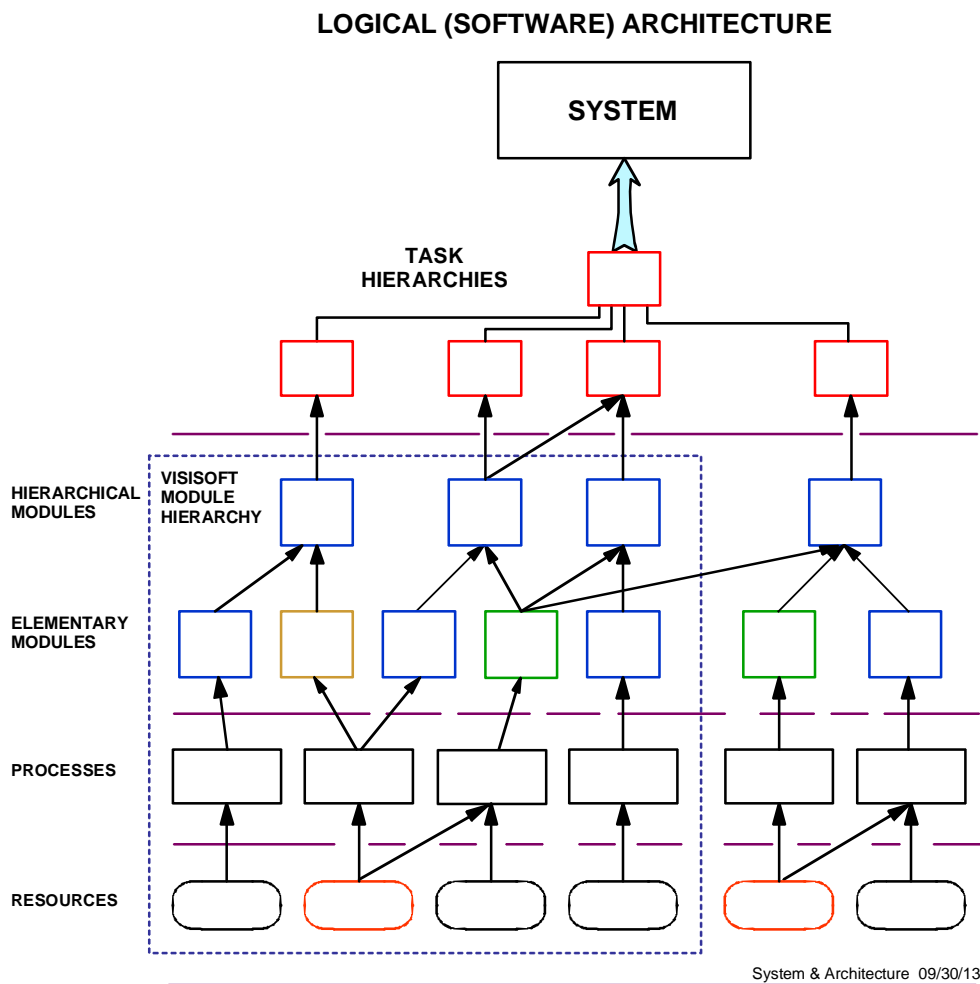


Figure 9-5. Overview of the logical system hierarchy.

Run-Time Software Systems

Top level architectures may contain run-time software that spans multiple computers, with each computer having its own OS, and typically using communication links between computers. On a given computer, multiple executable *tasks* (separate executables) may reside in one or more directories. By the definitions used here, a single computer may contain many processors operating under a Single OS (SOS). Thus, a run-time software system may span different types of platforms and operating systems.

When one task starts another task, the second task becomes part of an executable *task hierarchy* that resides on a single computer, refer to Figure 9-5. During execution, tasks that are part of a task hierarchy may attach to multiple *shared memory segments*. Such tasks are started by other tasks higher in the same hierarchy. When tasks are started by separate user actions or tasks outside the hierarchy, they are *independent*. Independent tasks (not part of a task hierarchy) may attach to *global memory segments* on the same computer.

Although multiple tasks can share memory easily in this CAD environment, our main concern here is with a single task containing multiple *threads* (sometimes referred to as p-threads). Threads may run concurrently on separate processors in a true parallel processor environment. i.e., running on multiple processors under a SOS, referred to here as a *SOS parallel processor*. This type of operation is described in subsequent chapters.

ARCHITECTURE ENVIRONMENT

Software Architecture

Using the CAD development environment, software architects decompose a system into modules by grouping resources and processes into *elementary modules* as shown in the Drawing Level 1 (layer 3) in Figure 9-2. *Hierarchical modules* are created by grouping modules into higher level modules as shown in the same figure. Figure 9-3 shows a library module that is sufficiently complex to warrant its own drawing. In general, modules are independent if they share no resources (i.e., they are not connected). Modules may be covered to hide detail.

One of the most significant observations drawn from this framework is the recognition that scope rule declarations are obscured at the language level. They are unnecessary in the languages that support this framework. It is the architecture (connected by lines) that determines how data is shared, and the corresponding independence of modules. Having developed an architecture, programmers can implement the data structures and rules using the resource and process languages. These may be edited directly as illustrated in Figure 9-4.

Unless one has witnessed directly the development of such architectures, the above discussion may take time to comprehend. Having used it, it is apparent that architecture as defined here is as critical to software design as it is to hardware design, with or without parallel processing. It is why productivity multipliers are very high when using this CAD system, especially in the support mode when a new person has to understand what another has created.

Visualization Of The Architecture

In the approach described here, architects decompose a system into large independent modules. If there is sufficient inherent parallelism in the system to warrant a parallel processor, the inherent parallelism in the system must be mapped into independent modules. Then threads must be contained within the independent modules. As described in later chapters, when threads in one module are independent of those in another, they can run concurrently on separate processors. When a module is assigned to a processor, all of the threads in that module go with it. Thus the distribution of threads onto parallel processors can be implicitly optimized by the architecture.

Module architectural information, including connectivity, is contained in databases that support the CAD development environment. A Run-Time System is then generated that uses this information to control OS calls that allocate processors to modules. It automatically ensures that the resources (data) reside with the processes (instructions) that use them. If there are enough processors to house each of the independent modules, load balancing becomes unnecessary, see [114].

Scalability

Increasing complexity may cause a software development effort to scale nonlinearly, i.e., the effort required to build a system increases faster than its complexity. This is characterized by Fred Brooks in *The Mythical Man-Month*, [19]. However, linearity depends upon independence. Linear scaling of complex systems can be achieved by maximizing module independence. This phenomenon is apparent from the separation principle.

As indicated above, languages provide limited control over the design of large complex software systems. Architects of industrial buildings, ships or airplanes would find it very difficult, if not intractable, to produce large complex designs without drawings. It is now apparent that software is no different. Having used the CAD interface and drawings to produce architectures, and having observed module independence by visual inspection, it becomes obvious that developing software without the architectural drawings described here is a great disadvantage. Visualization is critical to controlling increasing complexity when building and supporting software in general.

DESIGNING PARALLEL PROCESSOR ARCHITECTURES

Designing software architectures to maximize speed on parallel processors requires additional considerations and trade-offs. These are addressed below

Architectures Must Be Based Upon Inherent Parallelism Of The System

When translating application requirements into a software system to run on parallel processors (e.g., multi-core chips), one must take maximum advantage of the inherent parallelism in the application. This requires the knowledge of a *subject area expert*. Examples are simulations of molecular type structures affected by gravitational or electro-magnetic fields. Parallelism in an application must be translated into a corresponding software architecture (if each instruction depends upon the prior one, there is none). The system described here has been designed so that subject area experts can develop and experiment with different parallel software architectures directly.

High Processor Utilization Efficiency Through Maximum Processing Overlap

The percentage of useful processing overlap, i.e., processors running concurrently doing useful work, as opposed to waiting for inputs or overhead (e.g., inter-processor communications, swapping and paging), directly determines the speed multiplier achieved using parallel processors. It is common to obtain between 5% to 10% efficiency using current approaches. At 10% - considered good - one needs 200 processors to get a speed increase of 20. Using VisiSoft on a Parallel PC, one should expect 70% to 90% or greater efficiency (at 80%, a speed increase of 24 may be obtained from a 32 processor PC).

Applications with a reasonable degree of inherent parallelism (greater than 50% but not embarrassingly parallel) and heavily loaded scenarios produce the greatest speed multipliers. One can view speed improvements for these applications from two aspects. First is the use of more processors to increase the speed, e.g., using 10 times the number of processors to gain improvement. Second is when the speed constraint is fixed but the number of processors is minimized using the techniques described here. In the latter case, a nonlinear reduction can occur because the spatial footprint of the hardware can be reduced considerably, reducing inter-processor communication, transmission and memory boundary crossing delays. Using memory to stack shared data is a critical factor in increasing the overlap. Other factors include single processor speed improvements and architecture improvements from using VisiSoft, see [37].

Selecting Software Spaces To Simplify Design And Gain Speed

When using mathematics to represent complex physical systems, engineers have learned to select a coordinate system that simplifies the equations and maximizes speed. The property of linear independence of coordinates supports this criteria, even when the system itself is nonlinear. For example, when using the *State Space* framework to solve electronic circuit design problems with large state vectors, one must select the *state vector* that diagonalizes the matrix (minimizes off-diagonal terms), simplifying the equations and maximizing solution speed.

This approach applies directly to software design, especially for parallel processing. It requires expanding mathematical notation beyond that of numbers. As shown in [37], software spaces are best mapped into hierarchical databases. Using this concept, *Generalized State Space* is introduced as a mathematical framework to support the solution to software problems. Data spaces used to solve complex problems are *Generalized Spaces* or *State Vectors*. Software algorithms are *Generalized Transformations* on these spaces. Just as in typical mathematics, one must define the state vectors of a system before defining transformations that describe the dynamics. Design of the data spaces is key to simplifying the design of corresponding transformations (algorithms) performed by software.

INDEPENDENT (IND) MODULES

Given applications with a high degree of inherent parallelism running on efficient parallel computers, their effective use comes down to a few major factors. First is ensuring that full advantage is taken of the inherent parallelism in an application - a software architecture problem. Second is allowing subject area experts to describe their problem without having to twist it into special computer languages. Third is taking full advantage of an optimized architecture of IND modules to maximize run time overlap and therefore processor utilization efficiency.

Preparing Parallel Processor Tasks

When preparing a parallel processor task in the development environment, one must prepare each of the elements of the architecture in terms of the following hierarchy:

Top Level Task

- Independent IND Modules
 - Hierarchical Modules
 - Elementary Modules
- Processes
- Resources

IND modules must be mapped into memory by the VPOS Link Editor using a Module-ID as an “offset” to the memory addresses for the rest of the module. This allows IND modules to be loaded onto separate processors so as to optimize the proximity of the data memory to the instruction memory for that module. The addresses assigned by the Link Editor are relative to a physical offset address. In the case of a module requiring a huge amount of memory, i.e., one that is close to or exceeding the limit on local memory (e.g., 4 Gig), additional addressing facilities are needed to support the load step.

USING INSTANCED MODULES

Using the CAD system described here, users are able to define *instanced modules*, i.e., define the number of instances of a module and build instanced module hierarchies. This simplifies descriptions of both the module information structures and the module rule structures. It eliminates the need for pointers at the language level. Pointers are eliminated from both the module information structures and the module rule structures. Instances are declared at the architecture level and when specific instances are scheduled to run. Otherwise, there is no need to distinguish between module instances. By definition, all instances behave the same. What they do depends upon their individual state vectors at a particular instance of time. Specifically, the system provides for the following:

- Users define the *quantity of module instances* and the *name of the module instance pointer* in the architecture environment when creating or modifying a module.
- Every resource within the module is automatically translated into multiple independent instances (copies), one for each of the module instances.
- *Hierarchical instances* can be defined by declaring the different module instances at corresponding layers of the module hierarchy.

DEFINITIONS

The following definitions reflect the rules of the CAD approach used here.

Classes Of Modules And Their Elements

- **SHARED INTERFACES** - Modules that are connected by shared resources have *shared interfaces*. For example, two modules have a shared interface if a process inside one module shares a resource with a process inside another module. The shared resource is the shared interface.
- **INTERIOR AND INTERFACE ELEMENTS** - Processes (resources) are *interior* elements of an elementary module if they have no shared interfaces with resources (processes) outside that module. They are *interface* elements if they do have a shared interface. The interior elements of an elementary module are interior to any higher level module containing that elementary module.
- **INTERIOR AND INTERFACE MODULES** - Modules are *interior* to a hierarchical module if they contain no elements with shared interfaces outside that hierarchical module. They are *interface* modules of that hierarchical module if they have elements with shared interfaces outside that hierarchical module. Modules that are interior at a given level of a hierarchy are interior to all higher levels.
- **INDEPENDENT MODULES** - Two modules are defined to be Independent (IND) if they have no shared interfaces. The independence property implies that these modules may run concurrently and produce results that are complete and consistent with those produced when not running concurrently. Given a set of initial conditions within the resources of each, the results of each will be complete and consistent when any of their internal processes run.
- **INSTANCED MODULES** - Modules can be defined to have multiple *instances* at the architectural level. This implies that, at run time, *each instance* of the module must have an *independent* copy of every resource in the module, corresponding to *instanced resources*. Similarly, every instance contains a copy of each process in the module.
- **INSTANCED RESOURCES** - Resources are defined to have multiple *instances* when they are elements of an instanced module at the architectural level. This implies that, at run time, *each instance* of that resource exists as an *independent copy* of that resource and is referenced by a unique name determined from the resource name and instance number.
- **HIERARCHICAL INSTANCED MODULES** - Instanced modules may be defined within instanced modules hierarchically. Resources contained in the lowest level instanced module will have as many independent copies as the product of the successive instances in the hierarchy. Similarly, processes contained in the lowest level module will have as many independent copies as the product of the successive instances. Hierarchical Instanced Modules may be IND Modules.

Classes Of Independence

The following paragraphs define the property of independence as it applies to the various types of modules defined above.

- **SPATIAL INDEPENDENCE** - Two processes are *spatially independent (not connected)* if they share no resources (memory), independent of time. From here on in this section *independent* will imply spatially independent. Two modules are independent if every process in one is independent of every process in the other, i.e., they have no shared interfaces. Interior processes of an instanced module are independent from those of other instances of the same module. Module instances are independent if they have no shared interfaces. Interior module instances are independent.
- **TEMPORAL INDEPENDENCE** - Processes (modules) may be *independent in a given instance of time*, but dependent in another instance of time. If two processes are using the same instance-pointer value to reference a resource in an instanced module at the same time, then they are not independent at that time. However, if they reference instance-pointer values for that same resource at mutually exclusive times, they may be independent.
- **INDEPENDENT MODULES** - Modules that are independent *cannot* share resources, and therefore cannot have shared interfaces. If modules are designed to be independent so they can run on separate processors, and if those modules are to exchange information, they must do so through a special interface, where a copy of each interface resource resides in both modules. These resources must be connected through an Inter-Processor Communications (IPC) manager, used for sharing data across processors. The IPC manager ensures completeness and consistency between independent modules running on separate processors.
- **INDEPENDENT INSTANCED MODULES** - These are basically the same as independent modules, except that they have a hierarchy of instancing at the interface, so that modules can be communicating with more than one instance at a time. This is illustrated in the sections that follow.

Denoting Independent Modules In The Architecture

Independent Modules, including those that are instanced, are denoted using **VDE** when working with the architectural drawing. IND Modules are denoted as IND MODULES using the Module Type panel, as when creating or modifying Library or Utility modules in the VisiSoft development environment, see [67].

COMMUNICATING BETWEEN INDEPENDENT INSTANCED MODULES

As described above, many of the opportunities for inherent parallelism occur with instanced modules. This is based upon the assumption that *module instances are independent*. As described in the previous chapter, except for the possibility of one or two processes in special *interface* modules, processes in one instance cannot share resources with those in other instances. When building large system modules, the ratio of internal processes to interfaces is typically greater than 10:1. These opportunities for taking effective advantage of a parallel processor provide the motivation for automating the design approach to instanced modules. Before describing the detailed design for inter-processor time and space (memory) synchronization, we will investigate the facilities required to simplify the architecture of instanced modules and their interconnections.

Modeling Mobile Communication Nodes

The developers of the CAD facility described here have been modeling advanced military communication systems since 1982. As it turns out, the design of protocols for sharing information over a fast-moving battlefield with steep terrain and foliage is similar to designing communications between processors when modeling nonlinear dynamic systems. On the battlefield, all nodes are mobile. This is totally different from a mobile phone system where only the mobile phones move. These phones are tied into a fixed infrastructure of communication nodes that are stationary, making the protocol design problem comparatively simple.

Given that the architecture of a simulation is broken into many models that are partially (albeit highly) independent, they must communicate with each other as well as perform their normal tasks. This is analogous to the military communications problem. We will use radio communication examples to describe the instanced module design cases.

When using mobile radio communications in rough terrain, signals may be received or lost as radios move. Thus, one must model the Electro-Magnetic (EM) environment in between nodes that determines whether or not they can hear each other. The EM environment model is typically - by far - the most complicated model in such a simulation and accounts for most of the computational burden. Yet it sits in between the instanced models of the nodes. This requires an important architectural solution to take advantage of a parallel processor. The following examples describe the approach to that solution.

Figure 9-6 illustrates many of the design issues to be considered when modeling mobile communication nodes using instanced models on a parallel processor. If the RECEIVER in a RADIO model schedules the use of R_F_LINK in an R_F_LINK model, it must specify the SOURCE and DEST instances and thereby expect R_F_LINK to be tied to the correct instance of LINK_INFORMATION.

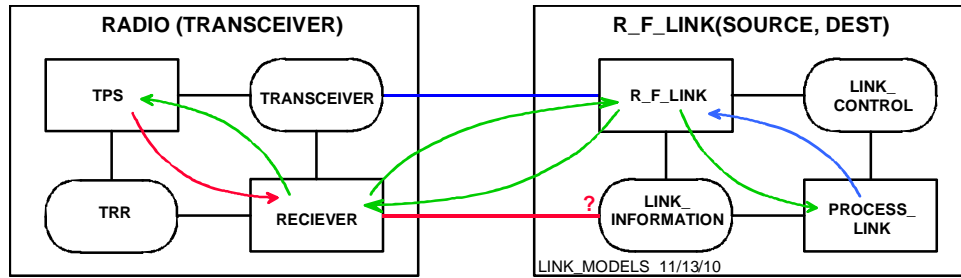


Figure 9-6. Example of an architecture with instanced resources.

Note that in this discussion we use a simplified notation that shows a model with a hierarchy of instances within parentheses, e.g., (SOURCE, DEST), even though this notation does not follow the CAD system. It implies that the instanced model must lie within a higher level instanced module and that the multiple instances are implicitly layered in the actual architecture. We also note that in communication simulations, the times associated with use of links and receivers are critical, often down to microseconds. A process that is *scheduled* corresponds to the start of a *thread* (a chain of process calls). Design of these models is critical in determining the correct physical outcomes. However, by carefully modeling the desired physical outcomes using schedule statements that state when processes are to be run using specified simulation clock times or delta-times, the corresponding timing and synchronization of threads will be invoked accordingly by the system.

In the above example, the instance of the RECEIVER process is known when the schedule statement is invoked. If RECEIVER schedules R_F_LINK, the correct instance of the TRANSCEIVER resource is automatically passed to this process.

Alternatively, if R_F_LINK schedules RECEIVER, it can specify the TRANSCEIVER instance. Since R_F_LINK is tied to a specified instance of LINK_INFORMATION, these instance pointers are passed to RECEIVER also.

If, however, TPS schedules RECEIVER, there is no way to know *automatically from the architecture* what instance of LINK_INFORMATION the RECEIVER process should be connected to. Furthermore, if any process outside the RADIO model attempts to schedule or call RECEIVER, the connection in red is invalid. Therefore, this connection cannot be allowed, rendering this form of architecture invalid.

Likewise, if any process outside the RADIO model attempts to SCHEDULE or CALL R_F_LINK, the connection in blue would be invalid. This is further explained in the next example.

Figure 9-7 presents a similar case when KP in MODEL_3 schedules LP in MODEL_2. In this case, there is no way to pass on the pointer *automatically* to resource TRS in MODEL_1. This implies that, if a resource is to be shared between MODEL_1 and MODEL_2 when LP is called from outside MODEL_1, that resource must reside within MODEL_2. Again, this connection in red cannot be allowed rendering this architecture invalid.

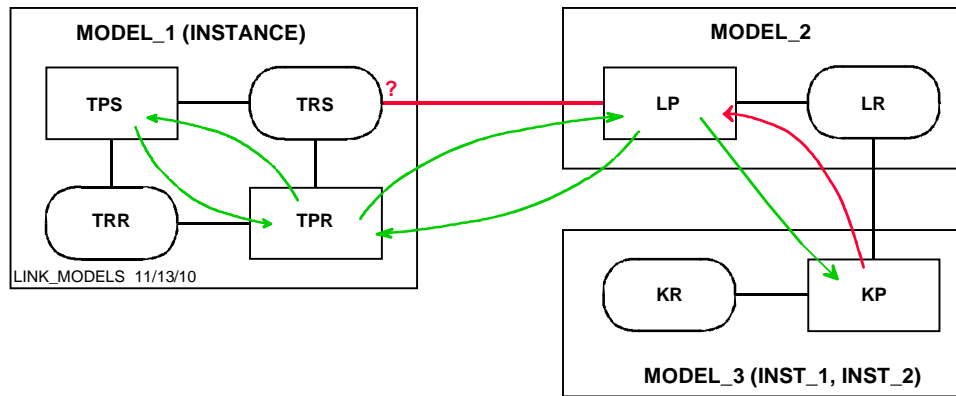


Figure 9-7. Example of another architecture with instanced resources.

There is a generic rule that applies to Figures 9-6 & 9-7. When a process connected to an instanced resource is scheduled, the instance pointers for that resource must be specified *based upon the architecture*, i.e., *explicitly* via the instance pointers in the schedule statement or *implicitly* based upon residence within an instanced model. In the case of the instance pointers, they must match an instanced model containing the resource; else the connection cannot be made architecturally.

CALCULATING RADIO CONNECTIVITY

One of the most common models encountered in communications system analysis is that used to represent a large number of radios or switches interconnected in a network. Switched systems are generally fixed in space, and their interconnections do not change with time, i.e., their connectivity is generally time-invariant. Radio systems are mobile, and their connectivity can vary significantly with time. The EM wave propagation calculations required to determine connectivity can take considerable processing time and are of particular interest here.

Figure 9-8 uses a radio model as an example. Each radio can have links to many others. A radio receiver can operate properly on only one link at a time, and the receiver model must account for potential interference coming from other radios that are transmitting at the same time. Therefore, each RADIO_MODEL must be connected to an ENV_MODEL that provides for all of the possible cross-link connections between radios.

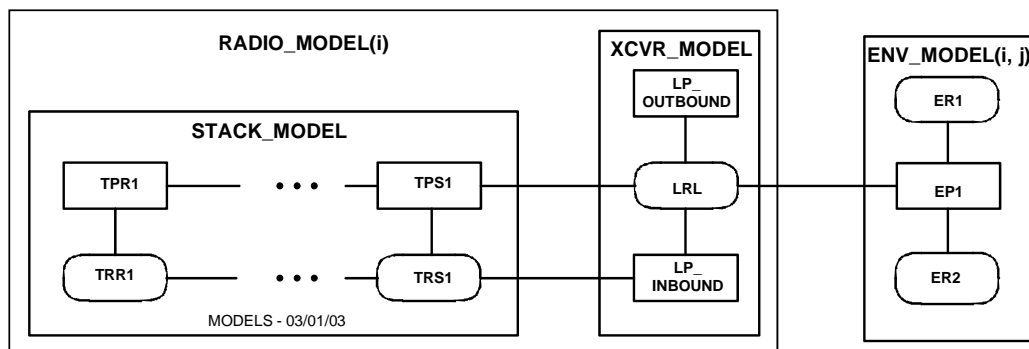


Figure 9-8. Example of good architecture for instanced resources.

The radio model in Figure 9-8 has an instance (i) for each radio. The environment model has an instance for each destination receiver (j) coupled with each source transmitter (i). All environment link instances (i, j) must be able to operate concurrently, just as each radio can operate concurrently. In the case of collision analysis, i.e., when two or more transmitters transmit to more than one receiver such that their signals sufficiently overlap in time, it is necessary for the model to have access to all link information at the same time.

When an instanced model interfaces with a non-instanced model, the non-instanced model can present a bottleneck that, depending upon the architecture, can be significant. This is the case with the radio environment model since it must determine the EM environment path loss, a significant set of calculations. Putting these calculations with each receiver removes them from the synchronization path as described below. When a model of a higher level instance interfaces with one of a lower instance, the case is similar. Again the solution will depend upon the architecture of the modules.

When source radio (i) transmits to destination radio (j), it does so through link (i, j). The environment link instance (i, j) gets scheduled from radio (i) to transmit a message to radio (j). Environment link instance (j, i) then schedules radio (j) to receive the message. For this to work correctly, the architecture must support process calls and schedules that automatically invoke the desired instance-pointers.

AUTOMATING INTER-PROCESSOR COMMUNICATIONS

For models (instances) to be independent, processes in one instance must not share any resources in another instance. Except for the interface resources in the ENV_MODELS, this is true for the architecture in Figure 9-9. With this architecture, each radio model instance can reside on a separate processor. Likewise, if desired, each environment model instance can reside on a separate processor. However, to minimize the time to cross-schedule threads on different processors, it is better that the environment model instances reside on the same processor as the corresponding radio model instance. This also eliminates the time to move data between processors. The resulting approach is shown in the architecture in Figure 9-9. There may be a trade-off between operating in parallel and operating sequentially. However, message transfer implies a degree of sequential processing between corresponding instances of affiliated models. This architecture may be best for a single processor as well.

Consider that radio instance (1) transmits a message to radio instances 2 and 4. This is accomplished by having instance (1) of process OUTBOUND_LINK scheduled with the message to go out. Since OUTBOUND_LINK(1) only interfaces with resources that are interior to RADIO_MODEL(1), the instance pointer is passed implicitly, so that the message is placed in LINK_DATA(1). OUTBOUND_LINK(1) then schedules EP11 as the transmitter for radio 1. EP11 uses LINK_DATA(1) to get the message and copies it to inter-processor resource ER11. This allows process EP11 to transfer data (using the IPC manager described below) from ER11 to the selected inter-processor receiver resources ER21, and ER41. It then schedules EP21, and EP41 on computers 2, and 4 to get the message.

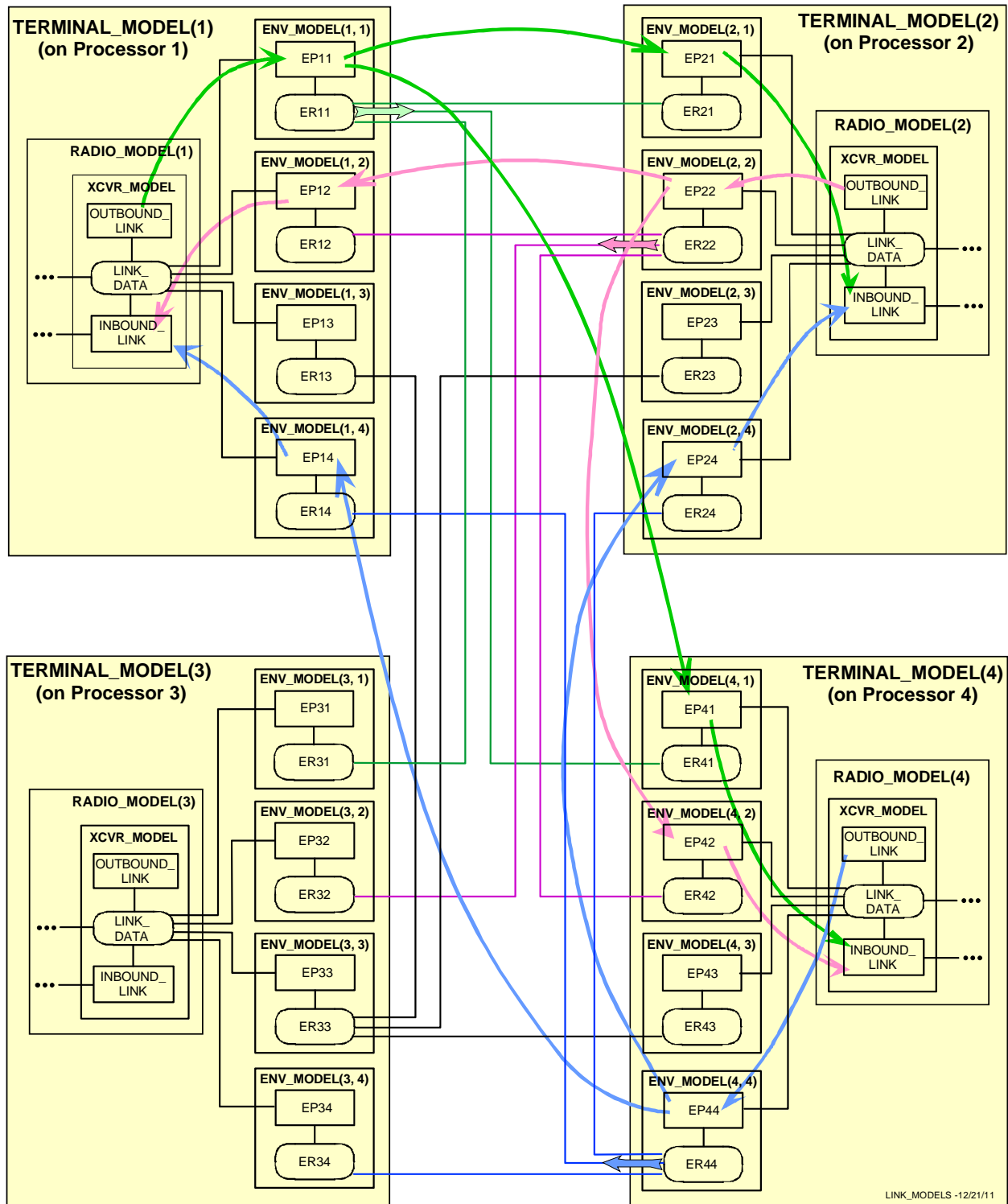


Figure 9-9. Illustration of a parallel architecture for instantiated models.

A symmetrically reversed series of events now takes place at `TERMINAL_MODELS 2` and `4`. In `TERMINAL_MODEL(2)`, `EP21` was scheduled by `EP11`. As the receiver of the message, `EP21` copies the message from the Inter-Processor (IP) resource and places it in `LINK_DATA(2)`. Since `INBOUND_LINK(2)` only interfaces with resources that are interior to `RADIO_MODEL(2)`, the instance pointer is passed implicitly. `EP21` then schedules `INBOUND_LINK(2)` to process the message.

Since the ENV models contain the inter-processor resources `ERij`, synchronization checks are processed automatically by the Inter-Processor Communications (IPC) subsystem in the Run Time System to ensure that the messages are received in the proper time sequence. The model designer merely schedules the transfers based upon the physical requirements, without concern for synchronization, race conditions, or consistency in general[†].

The sequence of events described above represents what typically occurs in a radio communication system, where most of the events are occurring concurrently with other events. We note that the transfer of messages from radio (i) to radio (j) and radio (k) may or may not be sequential. Many pairs of radios can be doing similar transfers concurrently, and this is where the inherent parallelism exists. This parallelism is best realized in a simulation if the model architecture follows the same physical design as the architecture of the real system.

We note that the above example supports analyses of the complex design requirements for the VisiSoft Run-Time System (RTS) (see Chapter 14). It is clearly *not* an embarrassingly parallel example. Ensuring that the design of the RTS supports all of the possible requirements in this type of discrete event simulation is key to producing one of the most complex elements of the parallel processor RTS.

Call Statement Rules

`CALL` statements are sequential; i.e., they cannot be used to increase the number of concurrent processes (parallel paths) or threads. They directly control any processes they invoke at the time, rendering them non-independent from the calling process. Calls from instanced modules will automatically carry the current value of the instance pointer(s) to the called process. If independent modules are to be run concurrently, they must be scheduled. *Called processes are in the same thread as the calling process.*

Calls that invoke a process on another processor, e.g., a utility, are not an efficient way to use multiple processors containing either the process or the call statement. As described above in the “False Memory Saving Dilemma”, multiple copies of frequently called utilities are a much more effective solution when they can run in parallel. This represents the typical time-memory tradeoff, with memory being relatively inexpensive, especially on a parallel machine. This leads to the requirement for utilities that can be copied (or instanced) to be distinguished from those that can't. More generally, *threads can not span multiple processors.*

[†] A property of the VisiSoft scheduler implying that processes scheduled at the same time and priority are ordered in sequence as they were scheduled.

CREATING INDEPENDENT (IND) MODULES FOR PARALLEL PROCESSORS

A major feature of the GSS architecture environment is the Independent (IND) Module illustrated in Figure 9-10. IND Modules IND_MAIN, IND_SAT, IND_F15_PLATFORMS, etc., may reside on different parallel processors under the single simulation GLOBAL_PLANNER.

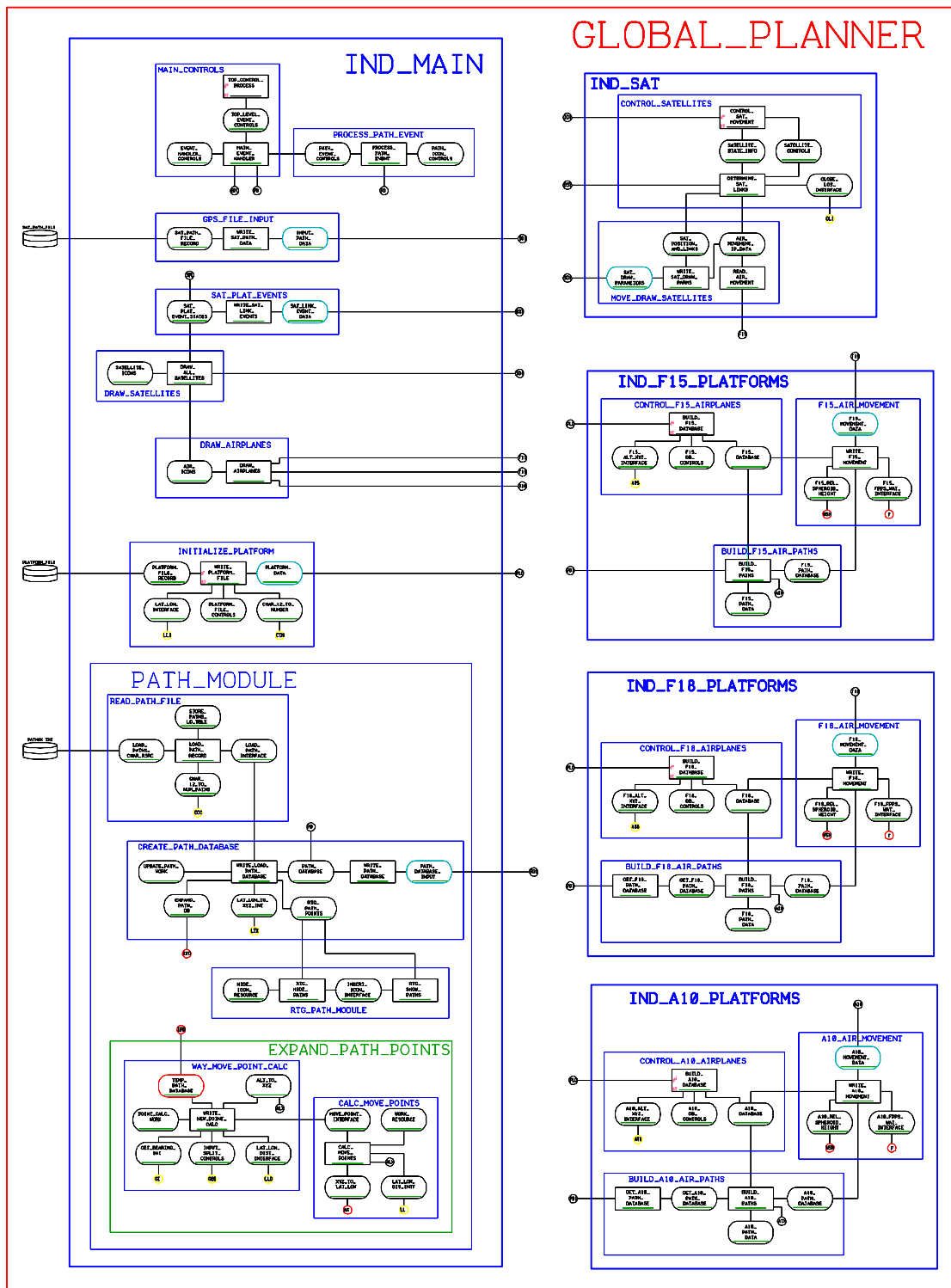


Figure 9-10. IND Modules in a parallel processor simulation.

IND Modules Must Be At The Top Of A Parallel Processor Task

The top level module(s) of a Parallel Processor Task (Simulation) must all be IND Modules. These IND Modules may also be instanced modules as shown in Figure 9-10. The number of instances, or IND Modules in general, is limited only by the memory in the machine.

In the case of fine grain models where the space is split into very small cells, e.g., those in particle or wave type simulations, one may increase the speed of the simulation by grouping multiple cells into separate IND Modules. This is not driven by the desire to reduce the number of model instances. It is driven by the speed of information exchange when dealing with huge numbers of processors, the resulting spatial footprint of the computer, and the corresponding potential distances between cells that must communicate. Algorithms that take full advantage of inherent parallelism may reduce the computational burden by preprocessing solutions that cover many cells, making them much faster than those running in parallel. The resulting grouped cells can take much greater advantage of the individual processors, being able to do more processing independently - in parallel - while minimizing the time delays between processors.

IND Modules Communicate Via IP Resources

IND Modules may only communicate with other IND Modules through shared IP Resources. In Figure 9-10, IND SAT and IND AIR_PLATFORMS each contain one IP Resource and IND MAIN contains four (green borders). Using VisiSoft, many processes may share IP Resources, but only one can have write privileges. These IP Resources, denoted by green ovals, must reside within the module containing the process that has write access.

IP Resources Support Full Duplex Communications

To communicate bi-directionally, in parallel, it follows from communications theory that, to ensure coherency, a full duplex approach must be used for two-way independent communications, implying the use of dual IP Resources. Figure 9-11 illustrates a combination of one-way write and read resources, one for each direction in a single module, IP_MODULE_11. A complimentary module is shown attached, i.e., IP_MODULE_22. These share a Full-Duplex communications channel. IP resources are automatically copied by to the colored shadow copies at the beginning of the receiving process (only one receiving process within an IND module may be attached to an IP resource). The shadow copies do not appear on the drawing. Because of the implementation of the IP Resources, coherency is ensured. Processes in separate IND Modules that are not connected to IP Resources may run concurrently on different processors.

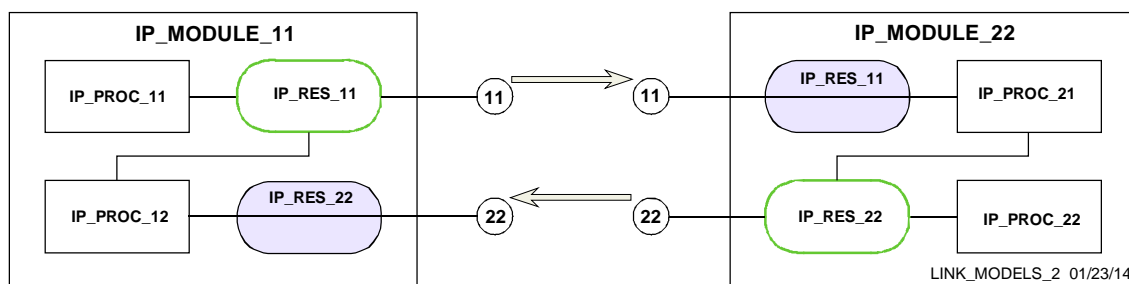


Figure 9-11. Example of *dual* paired One-Way IP Processes & Resources.

Figure 9-12 illustrates a one-way write module IP_MODULE_33. This module writes to multiple copies of IP_RES_33 in modules IP_MODULE_44 down to IP_MODULE_NN, a *one-to-many* case. The modules on the right can only *read* that resource, and the latest copy is made available to them when they start to run. In addition, IP_PROC_33 may RELEASE the resource at any time while it is running, and the receiving processes IP_PROC_41, etc., may ACCESS the latest copy at any time while they are running.

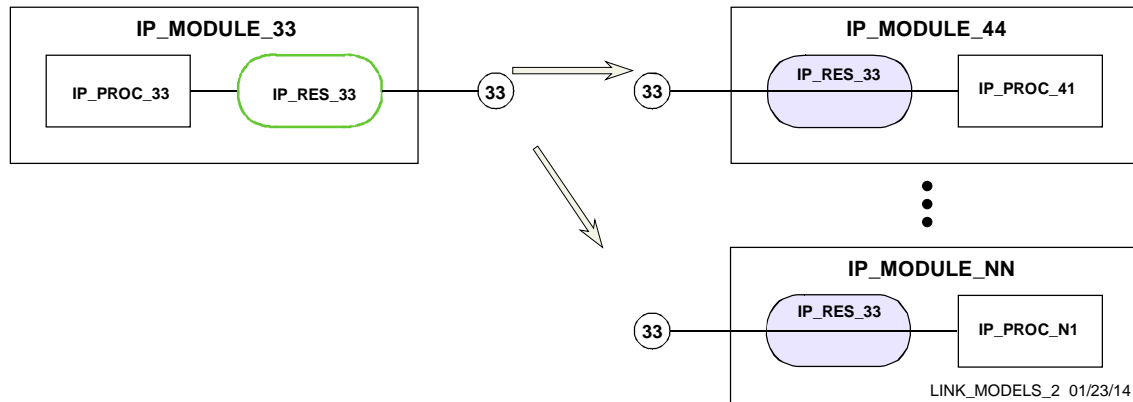


Figure 9-12. Example of single paired IP *one-to-many* Resources & Processes.

SAVING MEMORY - THE FALSE DILEMMA

Wasting Time By Saving Memory

In the case that one must model the processing of information from multiple instances concurrently, e.g., when doing calculations based upon signals from every radio, one could save memory by using a central utility or library module to be called from the environment model. These modules can store information on every link, including all of the databases required to do the processing. However, this approach is clearly a major bottleneck in that it may be called by many instances at the same time. In a light scenario, it may be used infrequently, e.g., only when there is movement or power changes in radios. This implies that, on the average, the links are stationary, occasionally requiring cross-link loss calculations for all links to a radio that has moved. In a heavy traffic scenario, i.e., those typically of interest, it will be used heavily.

Saving Time By Using Memory

The alternative approach is to copy the utility or library module onto every processor, including all of the databases required to handle every link. This allows the major computational burden for communications to be done in parallel, dramatically increasing the useful processor time overlap and thus the processor utilization efficiency as described in Chapter 6.

The decision to copy utility and library modules must be made by the model architect. Only an architect can characterize the inherent parallelism in the system being modeled (or built in the case of a software system). Such a decision should never be based upon the desire to save memory which is abundant in a well designed parallel processor.

One normally decides to use parallel processors to cut the time to obtain answers or keep up with real-time applications. Typically, time is of the essence. Parallel processor hardware designers who understand this trade-off have provided huge amounts of memory so that saving memory is almost never a concern. They have also addressed the problem of delay times crossing multiple memory boundaries. In the CAD system described here, this problem is addressed by co-location of resources with processes that use them. So, in general, an architect using a parallel processor should not be concerned about using huge amounts of memory.

Unfortunately, there are huge piles of software that exist in production where programmers have reused a single index number to mean different things in the same routine, saving 2 bytes while making it difficult to understand and potentially creating subtle bugs during maintenance. Worse, sharing data between modules (routines) to save memory is rampant in most modern development environments. Global memory does not exist in VisiSoft.

INTER-PROCESSOR (IP) COMMUNICATIONS

A general representation of the above architectures can be illustrated as shown in Figure 9-13 below. In this figure there are I instances of the top-level module MODULE_T, and J instances of the IP module MODULE_E within each of the MODULE_T modules. We note that in the radio example, it is likely that J is equal to or very close to I.

MODULE_T is instanced I times. Within each instance, MODULE_H is a hierarchical module that is independent of each of its counterparts in the other instances. Similarly, each instance of IPC MODULE_E within an instance of MODULE_T is independent of those in the other instances of MODULE_T.

It may be that only a few of the IP MODULE_E instances within a MODULE_T instance will be active at a given point in time during the course of a scenario. However, once a MODULE_T instance is assigned to a processor, all of the internal modules that have been previously called upon will be available when needed, without any additional assignments of threads to processors. When called upon they will just run - with their memory in place and ready to go.

INSTANCE POINTER VALUE RULES

To specify an instance from outside an instanced module, the instance value pointers (up to a maximum of $n = 6$) are assigned in a schedule or cancel statement. This implies that a hierarchy of instances within instances may not exceed 6. It should be noted that, historically, even 4 - the maximum ever used - is very rare. The format to schedule a process at the current time is as follows.

```
SCHEDULE process_name INSTANCE instance_pointer_1, ..., instance_pointer_n
```

When a process starts to execute, the instance pointers defined for modules containing that process hold the current values of the instances that the process represents. These instance pointers are used to automatically attach the proper resource instances to the process when it runs. Instance pointers are also available for use by the process in a *read-only* mode, i.e., values of module instance pointers cannot be changed by processes within that module instance.

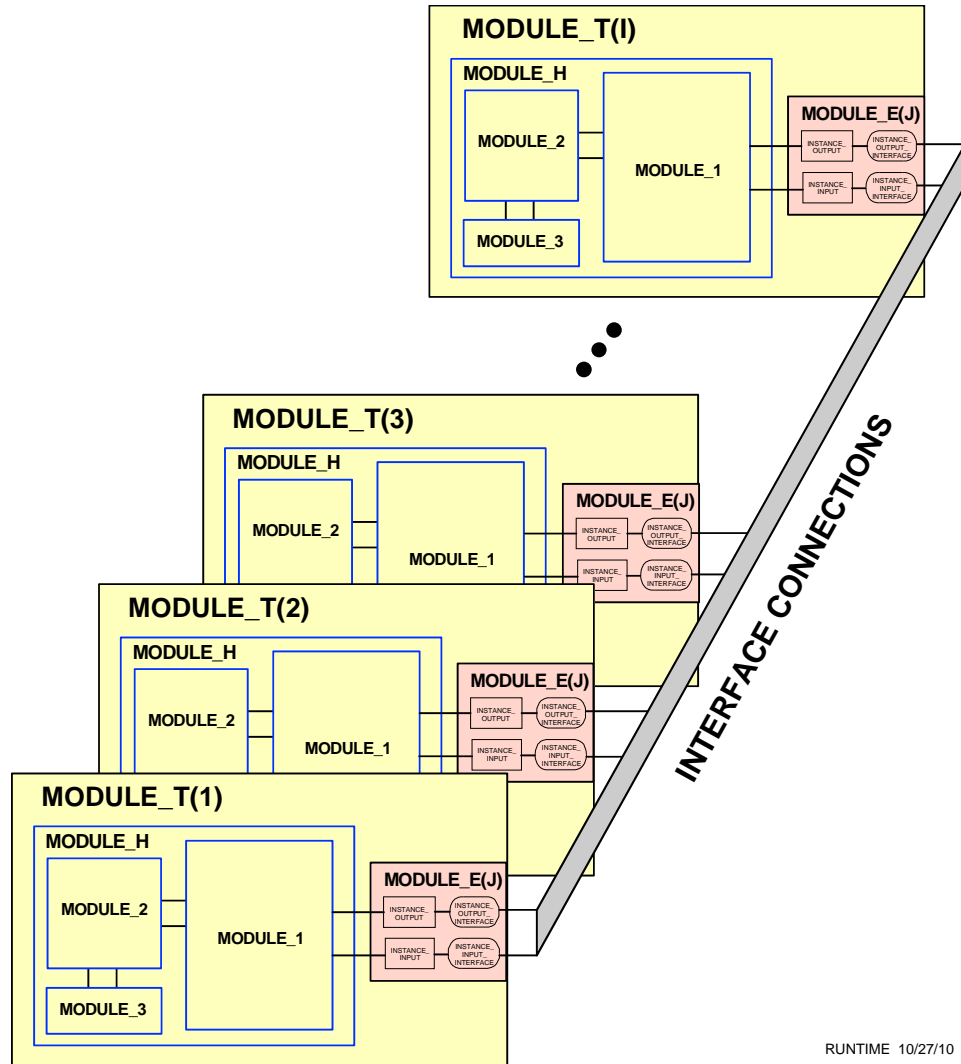


Figure 9-13. Example of a parallel architecture for IP resources.

When one process is scheduled by another in the same module instance, the instance pointers are passed implicitly and *must not appear in the argument list*. Because module instances must be independent, processes in an instance cannot schedule any in a different instance of the same module.

Referring to Figure 9-13, a process in MODULE_1 can schedule a process in MODULE_E with the pointer to the proper instance of MODULE_E being automatically invoked. However, the instance of MODULE_E must be explicitly provided:

SCHEDULE process_in_E INSTANCE j ...

where j may be any properly defined numeric attribute or literal.

The general format for a SCHEDULE statement is given in Chapter 12, Section 12.3.4.

MODULE INSTANCE CASES OF CONCERN

The following cases represent the allowed use of statements for transferring control among processes, including scheduling and deleting threads from the schedule. These rules are enforced automatically by the process translator in the development environment.

Case 1 SCHEDULEs, CANCELs & CALLs from a non-instanced module to an instanced module.

Referenced module (process) instances must be identified by specifying a value for the instance pointer, i.e., SCHEDULE/CALL process_name INSTANCE instance_pointer. CALLs may not occur across IND module boundaries.

Case 2(a) SCHEDULEs, CANCELs, & CALLs *within* the same module instance.

References to the instance pointers of processes within the same instance are implicit, being resolved automatically by the process translator and run-time monitor. Values of the instance pointers of a module are read-only by processes within that module.

Case 2(b) SCHEDULEs, CANCELs & CALLs *across* instances of the same module.

References to resources across instances of the same module must be accomplished by using a shared interface in a separate module. Direct resource references are not permitted across different instances of the same module. CALLs may not occur across IND module boundaries.

Case 3(a) SCHEDULEs, CANCELs, & CALLs from an instanced module to a noninstanced module.

The instance pointer of the referencing process is passed automatically to the referenced process by the run-time monitor. There is no need for an explicit reference to point back to the resource instances in the referencing module that the referenced process shares with it.

Case 3(b) SCHEDULEs, CANCELs & CALLs from one instanced module to another instanced module.

The designer must identify the referenced process instance by specifying a value for the instance pointer, i.e., SCHEDULE/CALL process_name INSTANCE instance_pointer. Again, pointers to resource instances within the referencing module that are shared with the referenced module are automatically passed to the referenced process. CALLs may not occur across IND module boundaries.

SUMMARY OF RULES

Single Processor Versus Parallel Processor Rules

There are essentially no differences in Resource or Process languages with respect to a single or parallel processor environment except for the CALL restrictions stated in the prior section. There are additional statements to support parallel processing in the Task Control Specification language. These provide for specification of a MAIN_MODULE and IND modules (they may be instanced) that may be allocated to separate processors by VPOS.

Creating And Addressing Instanced Module Resources

The instanced module quantity specification does not create multiple copies of instanced resources inside instanced modules directly. Instead, multiple copies are created automatically by the system translators and monitors. These are created as separate instanced data structures, with each effectively having their own names. Up to six levels of module instancing are allowed, including any QUANTITY levels of hierarchy within the lowest level module.

Basic Rules

The following general rules apply to both single and parallel processor environments.

- When a process in an instanced module is scheduled or called, the instance pointers must be specified explicitly if not implicitly. The values of the pointers are set as follows:
 - When referenced from a process inside the same module instance, the instance pointer is specified implicitly and must not appear in the instance_pointer list.
 - When referenced from a process outside the module, the module instance must be specified as an instance_pointer after the process name.
Example: SCHEDULE process_name INSTANCE instance_pointer
- When a process within an instanced module references another process in that same instance, it automatically invokes the same instance pointers. No arguments are specified relative to the common module instances after the process name.
- When referencing (1) hierarchical modules, or (2) other entities using multiple instance pointers, the pointers must be ordered as specified in the instance pointer list of the process being called. This must be in the order of (1) the hierarchy, from the top down, and (2) the instance pointers that do not reference module instances going last and ordered as specified in the instance pointer list of the called process.
- If a process within a hierarchically instanced module is scheduled from outside a subset of the instances, only the new instance pointers must appear in the instance pointer list of the process, in order from the top of the hierarchy down.
- Reuse of instance-pointer names in resources attached to any process interior to an instanced module must be qualified.

SPECIFYING A SOFTWARE OR SIMULATION TASK

A software system or simulation is defined by its control specification, of which many may reside in a given directory. Both of these are tasks when run on a parallel processor. Parallel processor tasks are composed of IND modules, each of which may contain hierarchies of sub-modules. If enough processors exist, each IND module may be placed on a separate processor. If not, multiple IND modules may be placed on a single processor. The number of processors to be used (assuming they exist) may be selected by the user. The system may decide on the allocation and assignment of actual processors to IND modules. Given that VisiSoft will likely use substantially fewer processors to improve speed over other approaches, processors assigned to a task will not be shared with other tasks.

Identifying The IND Modules In A Task

Each IND Module may have its own architectural drawing. Those IND Modules that are to be part of a task, must have their architectures reside in the same directory. [Downstream, VisiSoft may provide for IND Library Modules.] The current approach to identifying the IND modules to be used in a task will remain unchanged except that IND modules must be “top level” modules (not part of a module hierarchy). IND modules to be used in a task may be named in the control specification, and at least one of their constituent processes must also be identified with Start Codes in the architecture, along with their starting section and priority. IND Modules may also be started by Cross-Schedules and may be identified directly from the architecture and the corresponding “SCHEDULE/CALLing trees” maintained in the architecture database. The sections in which they are started (there may be more than one) will be determined by when they are called.

Parallel Processor Tasks

IND Modules must be defined in the control specification, along with the started processes, the sections in which they are to be started, and their corresponding priorities. On a single processor, the scheduler is used typically to support a discrete event simulation. However, that simulation may be tied to the real-time clock as part of a real-time control system, in which case it may look like a piece of real-time software.

In the case of software tasks residing on parallel processors, the IND Modules that may run concurrently on different processors must be synchronized. In this sense, there may be no implementation difference between a discrete event simulation and a software system.

It is important to note that the RESUME statement is used for resuming that part of a task that is suspended on a separate processor.

Software processes running concurrently will likely require synchronization. This may be achieved using the SCHEDULE statement, which ensures that certain processes run before others, and that they all synchronize at a given point. When using the scheduler, this point is defined by the ΔT_{max} interval, which is selected by the software system architect.

In the case of discrete time simulations of certain types of physical systems - sometimes referred to as fine-grain - running on parallel processors, they are typically broken into cells that may run concurrently. These cells communicate via “cell faces” that represent the interface data shared by the cells. Cells typically access data from their faces at the start of a clock period, and update the cell faces at the end of a clock period. In between the clock periods, they can run concurrently. Interchanges within a clock period as well as those used to terminate that period may be timed using the schedule statement. These schedules should be apparent from the application requirements and easily recognized by a subject area expert.

MEMORY SHARED BETWEEN IND MODULES

The manner in which memory is shared between IND modules within a task will depend upon whether these modules are loaded onto the same processor or different processors. In either case, this decision is made by VPOS. The user may designate that an IND module is to be run exclusively on a separate processor in which case memory will be shared accordingly.

When loaded onto the same processor, the threads inside IND modules must run sequentially, with synchronization guaranteed by the sequence of operations. When loaded on separate processors, memory shared between IND modules will be shared through the Inter-Processor Communications (IPC) system. These facilities eliminate concerns about data synchronization.

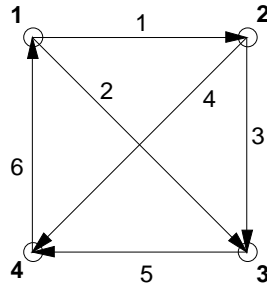
VSE & GSS TASK OR PROCESSOR EVENTS

When designing software using multiple tasks or multiple processors, events may occur in one task or processor that are required to trigger events in another task or processor. This may be done for communications or for synchronization. For example, when writing into a resource shared between two tasks or processors, one may want to determine if the write operation is completed on one processor before it is read on the other. This may be handled using the EVENT attribute. This attribute is a special VisiSoft data type that is processed by VPOS to ensure that EVENTS that occur in one task or IND Module are immediately processed by tasks or IND Modules on another processor. Changes in EVENT attributes in one task or IND Module typically cause a process to resume in a different task or IND Module on another processor, supporting synchronization requirements.

DETERMINING CONNECTIVITY BETWEEN IND MODULE PLATFORMS

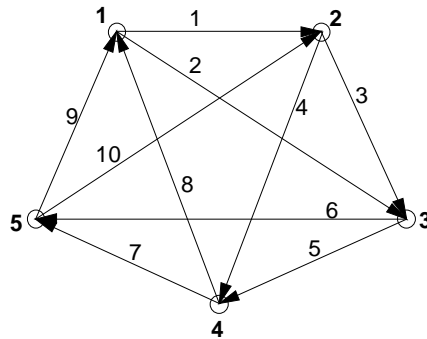
Figure 9-14 provides the basis for an analysis of the calculations required to determine connectivity between N objects (airplanes, satellites, etc.). The number of possible connections and the approach to minimizing the calculations is given in the figure.

$$N(N-1) / 2 = 4 \cdot 3 / 2 = 6$$



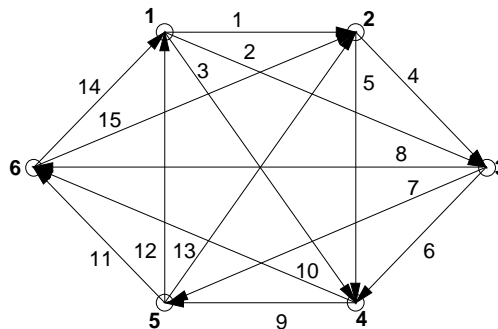
1: 1, 2 - 2: 3, 4 - 3: 4 - 4: 1

$$N(N-1) / 2 = 5 \cdot 4 / 2 = 10$$



1: 1, 2 - 2: 3, 4 - 3: 5, 6 - 4: 7, 8 - 5: 9, 10

$$N(N-1) / 2 = 6 \cdot 5 / 2 = 15$$



1: 1, 2, 3 - 2: 4, 5 - 3: 6, 7, 8 - 4: 9, 10 - 5: 11, 12, 13 - 6: 14, 15

Figure 9-14. Connectivity between 4, 5, and 6 objects.

As indicated in the examples in Figure 9-14, the number of possible connections between N objects is given by $N(N - 1)/2$. Since connectivity may be determined by either of the two objects, one is best spreading the calculations so they are done within each object on a separate parallel processor. Figure 9-14 contains the potential sequence of calculations below each of the connectivity drawings for each of the objects to spread the calculations.

Table 1 provides a look at the spreading of calculations when the number of objects ranges from 4 to 29. When N is odd, the number of connections is divisible by N, providing an even number of calculations for each object (right-most column). With 13 objects, there are 78 possible connections total, so that each object can do 6 calculations to determine the connectivity of the set. Note that each object does half of the calculations for connectivity with those that it is connected to. This is because connectivity only needs to be done in one direction for each pair.

Table 1. Optimal spreading of calculations.

N	$N(N - 1) / 2$	EVEN X
4	6	
5	10	2
6	15	
7	21	3
8	28	
9	36	4
10	45	
11	55	5
12	66	
13	78	6
14	91	
15	105	7
16	120	
17	136	8
18	153	
19	171	9
20	190	
21	210	10
22	231	
23	253	11
24	276	
25	300	12
26	325	
27	351	13
28	378	
29	406	14

Looking back at Figure 9-10, if 16 processors are used, three are required for the separate IND Modules: MASTER_SYNCHRONIZER, IND_MAIN, and IND_SAT. This leaves 13 processors for 13 IND Modules for airplanes and other platforms, whereby each platform would be required to do a maximum of 6 LOS calculations every time they move. If 5 processors were used for 5 platforms, each platform would only need to perform 2 calculations. Alternatively, using 5 processors for 15 platforms, one could put 3 platforms on each processor with each platform doing 6 calculations. Using 29 processors requires 14 calculations.

FINE GRAIN MODELING

An approach to fine-grain problems, e.g., molecular structure models, fluid flow models, and meteorological models, is shown in Figure 9-15. In these types of models, one is typically concerned with the dynamics of particles under the influence of fields, e.g., gravitational, electromagnetic, or temperature. Field forces typically depend upon distance from the source of the force, decreasing as $1/R^2$ in most cases. Forces on each particle may also depend upon those emanating from the other particles. These models are typically described by systems of partial differential equations in three dimensions as well as time, and the state vectors may be large, involving position, velocity, acceleration, etc.

When translating the system of equations into digital form for computer solution, one typically discretizes both time and the physical space. Time is represented by sufficiently small ΔT window increments while space is discretized into sufficiently small cells, both to represent continuous dynamics with sufficient accuracy. Typically one may have to use a parallel processor because of the huge computational requirements. Some applications allow approaches that place a single cell on each processor. The approach presented here is aimed at getting a potentially more accurate solution in a shorter amount of time with a much smaller number of processors.

In the example shown in Figure 9-15, minor cells are grouped into a $10 \times 10 \times 10$ structure of 1000 to create a major cell. The example then proceeds to use a $3 \times 3 \times 3$ structure of 27 major cells to model the system (only two 3×3 structures are depicted in the figure).

The intent is to show how a large number of cells can be grouped on a single processor with internal cells having no interfaces with the other major cells. Using this approach, there are only 300 external interfaces to the other major cells out of a total of 3000 interfaces. Each cell must take the information from the interface to the adjacent cells (6 each) and the central information from within that cell, and perform the calculations to determine the new central and interface values for the next time step. However, there need only be a single resource to support all of the internal cell interfaces, and only a single resource for each of the faces (max of 6) for the external cell interfaces.

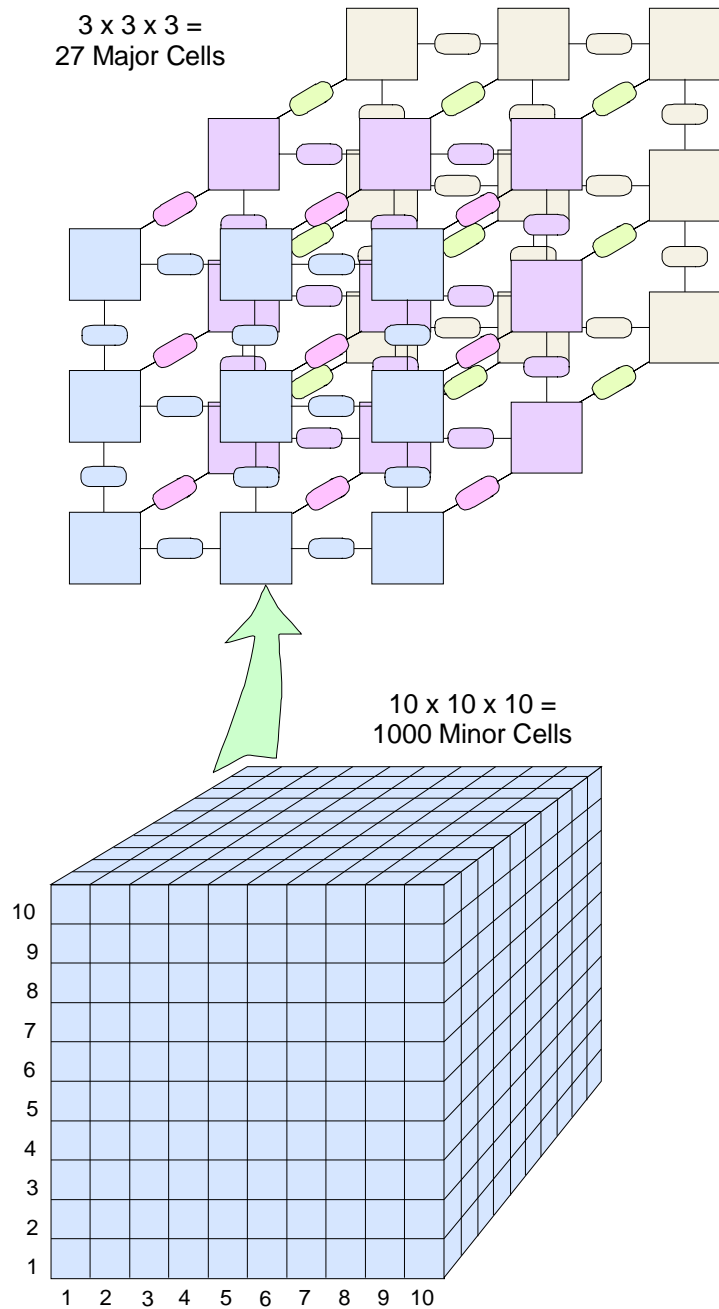
Using this approach, if the system is linear, each major cell is doing significant calculations that are independent of the other major cells. If the system is nonlinear, then one can estimate the values for the nonlinear interface as a starting point and iterate if necessary using a fast converging linear segmentation algorithm that may be used for all the cells. In either case, the amount of computation within each major cell will be huge compared to the cross processing necessary to update and synchronize the interface processing.

The result is that the interface processing between major cells will be a small part of that done within each major cell. If IND modules are then grouped on processors so that the processor utilization is approximately the same on each, a high Processor Utilization Efficiency (PUE) will result as described in Chapter 6. One may then expect to get a speed multiplier that will be a high percentage of the number of processors housing each major cell (27 in this case).

For many problems of this type, one can gain further increases in speed using optimal sparse matrix techniques within each major cell, making those computations significantly faster (typically a factor of N where N is the side of the $N \times N$ matrix).

A matrix of 27 major cells
 Contains 27,000 minor cells.
 If each major cell is an IND Module
 then 27 processors contain 27000 cells.

A single resource is shared between
 the adjacent face of each major cell.



A single resource is shared between
 each face of each minor cell.

3D_CELLS 5/09/14

Figure 9-15. Approach to fine-grain models.

Similarity To Tiling

Fine grain modeling as described above is similar to tiling in FORTRAN or C-based languages where tile directives are used to group cells into tiles. This is generally accomplished by grouping (tiling) multiple iterations within a 3D loop structure. Tiles are then assigned to processors to increase the workload on each processor while reducing the number of processors required. In most of these applications, the nature of the algorithms is not substantially changed from cell to cell, and the interfaces are typically of the same structure. Figure 9-16 illustrates a more general facility using VisiSoft IND modules that may perform different functions.

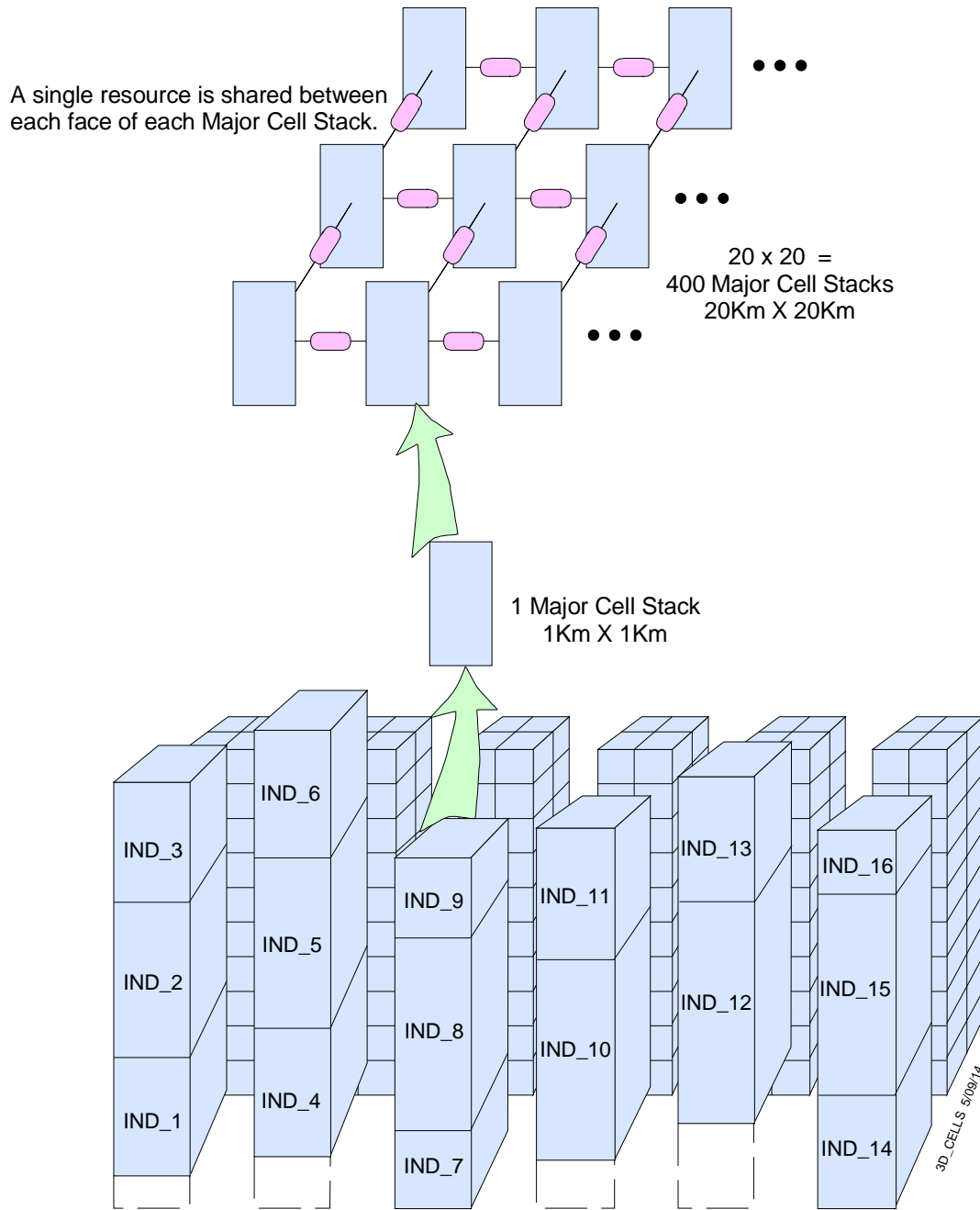


Figure 9-16. Grouping minor cells into IND modules for placement onto separate processors.

Using Discrete Spaces - A More General Approach

In applications covered in Chapter 19, IND modules can be totally different with different sets of instances. As illustrated in Figure 9-17, multiple IND modules may be best designed using multiple spaces. This allows one to represent different shapes that interface at similar surfaces, where each shape is best represented in a different space. As shown above in Figure 9-16, each space that interfaces with another can be connected by a different IP resource. This simplifies modeling of fluid flow type systems where constraint surfaces can be modeled using the best space for each of the particular shapes that are connected.

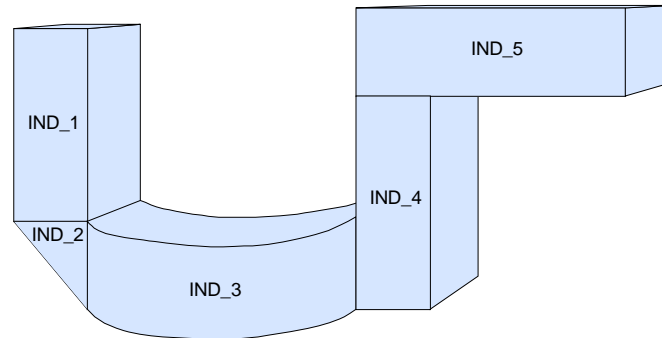


Figure 9-17. An example of different spaces for IND modules.

These IND module spaces can also be “grouped” onto processors based upon their connectivity as well as the loading. This reduces the idle time on a given processor, increasing the PUE as described above. It also reduces the number of processors required to perform an application task.

When dealing with more general applications, module loading may be nonstationary. This implies that their processing intensity or connectivity may vary unpredictably with time. It may be desirable to reassign these modules to different processors during the course of a run. IND modules may also be a nonlinear function of the state vector, including elements in IP resources, thus requiring iterative solutions. These considerations affect the architectural design.

SUMMARY OF PARALLELISM, ARCHITECTURE AND DECOMPOSITION

Parallelism implies that a software architecture can be produced that decomposes the system into modules such that modules can be designated as independent, implying that they may run concurrently with other modules in the system when they are invoked. The decision to invoke a module at run-time is based upon application requirements and the software design. We note that the software architecture for a single processor may be different from that for a parallel processor. For example, copies of memory in each module may be used to gain speed in a parallel processor, whereas using more memory may cause more paging on a single processor. Such factors must be considered to compare speeds fairly, see [7].

The *order* of a decomposition is equal to the number of INDependent (IND) modules. Decomposition of order M implies that the architecture has decomposed the application into M IND modules that can run concurrently with each other.

When using n processors in parallel, the *speed multiplier*, S_n , is the ratio of measured values of run-time on a single processor (R_1), and run-time on n processors (R_n):

$$S_n = \frac{R_1}{R_n}$$

The percent *parallelism*, P , achieved by a software architecture running on a machine with up to N processors is measured as follows.

$$P = \text{MAX}_{n=1 \rightarrow N} \left[\frac{S_n}{n} \right]$$

We note that these measures must be obtained by experiment, and are independent of whether the software or hardware architecture is the same for both the single processor case and the parallel processor case. If minimum run-time for a single processor is achieved using a different software architecture than that for the parallel processor case, then different architectures should be used to produce a fair value for the speed multiplier. Likewise, one may use different hardware architectures; dependence of the speed multiplier upon hardware is described elsewhere, [39]. As indicated above, these outcomes may be scenario dependent, requiring measures of statistical distributions for validity. We can now define the *inherent parallelism* of a system (and scenario) as the maximum percent parallelism, P_{MAX} , which can be achieved theoretically by the best hardware and software architectures for both the single and parallel processor cases. We now describe ways to estimate these measures based upon models of hardware and software architectures and factors that affect them.

Estimating Speed Multipliers

Given a software architecture that decomposes a system into IND modules, then a module may take the same amount of useful processor time on a separate processor as it would running in the single processor case. However, if the hardware or software architecture in the parallel processor case is different from that of the single processor, then the useful running time will likely change. For example, a table may be used to store information for hundreds of instances of an entity being simulated on a single processor and common data for each instance may occur once. On a parallel processor, the data for each instance may be on a separate processor, eliminating the table, but the common data will be repeated with each instance.

If the table is large, then paging may be required across multiple memory boundaries on a single processor, but not on a parallel processor since the table size may be cut by the number of instances. Common data used to manage a table is usually small compared to table sizes, but that also depends on the application. The point is that a decomposition that minimizes the run-time for a single processor will likely differ from that for a parallel processor. Also, cache sizes local to a processor may be the same for each parallel processor as for a single processor. Or one may select a single processor with much greater cache sizes to reduce the swapping and paging overhead from that of one of the parallel processors. Regardless of the differences in hardware, given that communication between processors is treated separately, swapping and paging within a processor on a parallel machine may be reduced dramatically from that on a single processor.

Overhead time may change also. For example, when modules on different processors are communicating, overhead will accrue that is not needed on a single processor. In addition, excess overhead time may accrue as well as idle time waiting for return communications. This time will effectively reduce the measure of parallelism. However, if a module requires a large database, and that database can reside in the cache of its designated processor with no paging, then overhead time may be reduced when it runs on a separate processor.

For each IND module, one must estimate the useful time, overhead time, and idle time for both the single processor case and the N processor case - where N equals the number of IND modules. This may be accomplished by estimating the single processor case, and then treating the parallel processor case as a normalized percentage (increase/decrease) of the single processor case. Communication between IND modules on different processors will accrue additional overhead as well as idle time in the parallel processor case. Thus overhead must be broken into parts: (1) time required for communication between processors, T_C ; (2) time required for swapping, T_S ; and (3) time required for paging, T_P . For the m^{th} module, total runtime overhead is given by

$$T_{OHm} = T_{Cm} + T_{Sm} + T_{Pm}$$

Both communication between processors and paging depend upon the software architecture as well as the parallel processor facilities. Communications will depend upon the size of data transfers, and the memory boundary crossing delays between processors. Paging will depend upon the memory size required for each IND module, the amount of local cache next to each processor, and the delays associated with each memory boundary crossing.

In the parallel processor case, if the number of processors equals the number of IND modules, then swapping overhead may be reduced to zero if the module remains by itself on that processor. This requires a run-time system that has the information on module independence so that all threads in an IND module are put on the same processor. It also implies OS facilities for controlling processor allocation and assignment from the task.

With a good software architecture, paging overhead for a module on a parallel processor may be significantly lower than that on a single processor if not nil. If there is a high degree of inherent parallelism, and a good software architecture to take advantage of it, it is conceivable that the speed multiplier could be greater than the number of processors. This would result from excessive swapping and paging overhead incurred on a single processor that disappears when a large number of IND modules are spread onto separate parallel processors. To achieve this also requires that the software architecture is taken advantage of at run-time.

The major factor affecting the speed multiplier is the ability to achieve overlap of useful times as illustrated in Figure 6-2. To estimate this, one must effectively estimate the useful time overlap between each IND module. This is not a simple process. One must consider that an overlap of 50% of the modules during a given time period represents 50% parallelism being taken advantage of during that time period. This must be done for all time periods. It is possible to capture this data by instrumenting the software in a way that is virtually nonintrusive.

To estimate the run-time for a single processor, (R_{SE}), useful time and overhead must be summed for each module, accounting for swapping and paging overhead.

$$R_{SE} = \sum_{m=1}^M [T_{Um} + T_{OHm}] , \quad \text{where } m \text{ represents the } m^{\text{th}} \text{ module.}$$

For the parallel processor case, the estimated run-time, (R_{PE}), will equal that of the module with the largest sum of useful time, overhead, and idle time.

$$R_{PE} = \max_{m=1 \rightarrow M} [T_{Um} + T_{OHm} + T_{Im}]$$

Since idle time and overhead may play significant roles, one must perform this estimate for each IND module to determine the total time. Modern parallel processors collect data that can help to estimate these times. Without such data, estimates may be very difficult if not intractable. What is important is to analyze this data to understand directions for improving architectures.

We note that the above approach is based upon the unsaturated case (number of processors available equals or exceeds number of IND modules). The saturated case is much more complex, depending upon load balancing as well as having IND modules share the same processor, and is beyond the scope of this treatment.

GENERAL SUMMARY

The speed multipliers to be gained from using a parallel processor depend upon a number of factors that may be evaluated prior to making an implementation investment. First is the inherent parallelism of system. This determines the potential for useful processing to occur concurrently (useful time overlap) on a parallel processor. Second, the hardware architecture must support the ability to access memory local to each processor concurrently as well as minimize the time for transfers between processors. Third, the OS supporting the hardware must provide facilities to allocate and assign processor resources by a run-time system that has been optimized to use knowledge of the software architecture.

Given that the above criteria are met, the most difficult hurdles to date have been the ability to: effectively decompose a software application with inherent parallelism; and effectively use the knowledge of that decomposition at run-time. This requires a software development environment that makes it easy to develop architectures of IND modules that minimize communication between them, reducing overhead and idle time. It also requires that a run-time environment be generated with knowledge of the module independence (the software architecture), so that IND modules are allocated to processors, and assigned in a way that takes maximum advantage of parallel processor resources.

Designing System Architectures

Visualization of architecture is critical to the design of an engineering system. It permeates the ability to address and control complexity, from decomposition to fault isolation. When looking at drawings that are many layers deep, it is apparent that orders of magnitude of complexity can be put under tight control through visualization. Where pertinent, these drawings reference clearly identified specifications.

In the Computer-Aided Design (CAD) approach presented here, data is separated from instructions using two separate language translators, one for Resources and one for Processes. One can click on the lowest level iconic elements in the drawing to see the “code”. The Resource and Process code is easily understood because of its hierarchical nature and English-like language. The code can be modified right on the drawing.

Modern engineering environments depend upon CAD systems to simplify the design and enhancement process. Huge sets of engineering drawings are automatically cross-referenced in large databases. Libraries of modules are stored for reuse in multiple drawings, and easily modified and identified accordingly. In many cases, these CAD systems provide simulation tools to further simplify and automate the design and test process.

System complexity is conquered when it is decomposed into an architecture of modules that are maximally independent and easy to modify. If new functionality is required, then both the architecture and design details must be easily understood to allow for redesign by other engineers. These are the same principles that simplify complex software in general and parallel processing in particular.

Approaches to this type of engineering design include the ability to:

- Decompose a system into modules and submodules that can be designed and tested independently - by different design teams.
- Design modules that can be redesigned and replaced with minimal impact on the rest of the system.
- Design modules that degrade gradually, support rapid fault isolation, and produce information regarding faulty components.
- Design modules that may be run concurrently (e.g., on a parallel processor).
- Design an architecture that supports control over the growing complexity of modules of a system as it is enhanced.
- Design a new sub-architecture so that further expansion is accommodated easily.

Reducing the cost of fabrication or implementation is part of the engineering design process. Ambiguities in either the drawings or specifications that result in improper implementation or fabrication are the fault of the engineer who did the design. Except for mathematical formulations, engineering specifications are written in English or other native languages for clarity, and accompanied by more drawings and pictures to ensure ease of understanding. Engineers are measured based upon the understandability of their design. If an engineer’s peers find it easy to understand, it is considered a good design.

THE CAD INTERFACE FOR SOFTWARE ARCHITECTURE

Because the underlying theme of this book is the theory of software, we have not included descriptions of the CAD interface and user facilities for building software architectures. Detailed descriptions of the user interface for this system can be found in the GSS or VSE User Reference Manuals, see [67] or [150]. Conversely, the elements of the languages that support this system are closely tied to the theory and are explained in detail in the following sections.

CHAPTER 10

SOFTWARE LANGUAGES FOR PARALLEL PROCESSING

We estimate that, in a competitive software product organization, 65% to 85% of development time is spent working with existing software modules. Therefore, the ease with which one can understand and modify an existing module directly impacts software productivity. This impact is amplified when changes are made by someone other than the original author. In most applications, one is also concerned about run-time speed. This chapter analyzes language properties that affect both productivity and run-time speed.

Conventional software languages require textual documentation - both external and internal (comments in the code) - for people other than the original author to understand complex algorithms. In most programming environments, enforcing production of textual documentation is difficult. Understandable code reduces the need for textual documentation as well as the time spent trying to understand complex algorithms written by another author. This chapter examines the properties of programming languages that affect the understandability of production code, and the approach to language design that ensures these properties. It also describes the properties of a language that support speed on a single processor, particularly the ability to easily create and manipulate complex data structures where a single move can take the place of many instructions.

Figure 8-2 depicts the interplay of external documentation, architecture, and language for the CAD software environment described in the previous chapters. Three separate languages are shown at the bottom of the figure: one for declaring data, one for stating instructions, and one for run-time control. These languages were developed and refined over many years specifically to maximize both productivity and run-time speed.

It should be apparent by now that the CAD approach described here is a complete departure from conventional software development using C-based programming languages and FORTRAN. Although this departure is most evident from the architecture environment, it is also true in the language environment. As shown in Figure 8-2, there are three languages, each with their own translators. Each translator is tightly tied to the Run-Time System environment as well as the architecture environment.

Why three languages? First, resources (hierarchical data structures) are created and modified separately from processes (hierarchical rule structures). This is the result of the Separation Principle, separating data from instructions so that the independence of modules may be determined directly from visualization using engineering drawings. As we will show, this provides for multipliers on run-time speed as well as on productivity during development. In addition, there are many properties of a run-time task that must be defined, such as use of specific files, hardware devices, library facilities, graphical facilities, and other controls, some specific to simulation. These facilities warrant a task control specification language that is both hardware and OS independent. When this is achieved, a complete software system may be developed on one set of computers and run on another set without change. This is the case with the CAD system described here.

GENERAL OBSERVATIONS ON LANGUAGES

Computer languages versus human languages

The Instruction Set Architecture (ISA) defines the language of the computer, i.e., the instructions. This language is specified in binary. However, the description is normally augmented using mnemonic codes or an assembly type language so humans can more easily understand what is represented by the strings of 1's and 0's.

In the early days of computers, engineers wrote programs in machine language using binary coding sheets. This was time consuming, error prone, and difficult to understand. The next round of improvement provided mnemonic codes for the instruction types, and decimal numbers for specifying memory locations. Although much more readable, it took many lines of code to do simple arithmetic, including the tracking of scale factors for real numbers.

As the need for a more human oriented language became apparent, assemblers came into being. This provided for relative addressing so that labels could be used to define memory locations for both instructions and data without being tied to specific hard-coded memory locations. Subroutines could be defined relative to a starting memory location and relocated to a different area of memory using relative addressing. Arrays could be accessed using indirect addressing, implying a starting location and an offset. Then floating point arithmetic became part of the assembler, so that scale factors were kept automatically. These human oriented functions were huge improvements for programmer productivity.

Fast forwarding ahead to FORTRAN and COBOL, these higher order language compilers quickly took over the translation from human-to-computer languages. People stopped thinking in computer languages and instead thought in terms of a much more organized design approach using higher-level languages. Programs were much more easily shared by humans. Because of the organized designs, programs ran much faster.

Fast forwarding again to CAD systems, these provided significant breakthroughs in productivity. They not only provided the human-to-human languages specific to an application area, they added graphical interfaces to help engineers easily create and change complex designs. These facilities minimized time consuming mistakes in human-to-human understanding.

Organizing Data Memory Resources For Speed And Understandability

The basic concepts of using Generalized State Vectors to maximize speed and simplicity of transformations leads to creating, changing and sharing hierarchical data structures. These data structures can be designated to serve different requirements, e.g., the following:

- Dedicated Resources - dedicated to a given process

- Shared Resources - shared directly between two or more processes

- Aliased & Shared-As Resources - shared by pointer between two or more processes

- Inter-Task Resources - global & local sharing data among multiple tasks

- Inter-Processor Resources - memory shared between processes on different processors

DRAWING BOUNDARY LINES AROUND THE LANGUAGE

In the case of the VisiSoft integrated approach, the architecture environment is used to produce software using easily understood architectures that provide maximum independence between modules. Therefore, the language must be designed to interface with the architecture facilities in the development environment.

On the other side of the VisiSoft integrated environment lays the run-time environment. This includes the Run-Time System (RTS), the VisiSoft Parallel OS (VPOS) and the hardware (see Figure 8-1). The language environment is used to generate a tailored RTS that, in turn, must interface with the VPOS and take maximum advantage of the particular hardware used to run the application. To take maximum advantage of a parallel processor, these interface design constraints have directly affected the language design.

Maximizing Speed

The main driver in language design is time: time to run, time to produce, and time to enhance. By produce, we imply a highly reliable piece of software; similarly for enhancements.

When the issue of speed comes up, some people think that writing in assembly language is more efficient being closer to the machine. Carrying that argument a step further, writing in 1s and 0s should be even more efficient. Both of these claims have been proven to be false in many experiments. Matrix inversion is probably an excellent example to observe. Writing in 1s and 0s or assembly language does not affect the speed of matrix inversion. Rather, it is a matter of mapping complex algorithms into an effective solution space. Visualization of this solution space is critical to understanding the concepts as well as designing the algorithms.

Underlying this problem is that of laying out complex hierarchical data structures that represent the solution space. These data structures are difficult to observe and work with in C-based languages as well as assembly language. But one must go through the experiment of building such algorithms in these languages. One then quickly observes the huge differences in understandability, as well as the sheer size of the resulting code. More importantly, it is difficult to think at the conceptual level without a language conducive to working at that level.

Using VisiSoft, all data is shared automatically by pointer. There is no global data. Data structures should be limited to what is needed to be shared between processes. The more processes that share a resource, the more likely a bottleneck will occur when using parallel processors.

Maximizing Understandability (By Whom Or What)

When we measure understandability, we are talking about measuring the time it takes for people - other than the original author - to understand the software. The C language was designed to make the compiler simple to write. The language is concerned with ease of translation into machine language, i.e., the language that the machine understands. C meets this requirement - translation to machine code is simple, making compilers easy to write. This goal is made clear in various Bell System/Laboratories journals. C was designed to quickly port a game onto a PDP-7 computer with a very small memory. Furthermore, the key developer of C liked Spartan syntax - read minimal keystrokes.

But in a large team environment, where the team is charged with the development of large pieces of complex software, where should the burden be? On the compiler or the programmer? Or, on the team trying to work together to build the application?

In the case of VisiSoft, the burden of language understandability is on the CAD system designed to support team building the total integrated development / run-time environment. It should be apparent from this book, that VisiSoft is not simply another compiler development effort. The background of the team that created it lays in the development of highly sophisticated CAD products for engineering design. The intent is to minimize the effort required to create complex software designs that are highly reliable and run extremely fast.

Using A Language To Ensure Understandability

Understandability clearly depends upon those using the language. And using the language clearly depends upon the facilities it provides. The topics below consider both sides of this issue.

Selecting Names

Very few highly experienced people responsible for large critical projects go back and write about their experiences. Jerry Sitner is a rare case, see [136]. In this reference he discusses selecting names of objects in a complex piece of software. When reading code representing an algorithm that controls many different objects, where each object has different numbers of instances, using I, J, and K provides no indication of what the object is whose instance is being referenced. Consider that the path database in Figure 10-1 represents paths for different types of platforms, e.g., airplanes, ships and vehicles on the ground. Consider also that there are different types of each platform, e.g., F15s, and F18s.

PATH_DATABASE			
1	PATH_RECORD	QUANTITY (500000)	
2	PATH_NUMBER		INTEGER
2	PATH_POINT		INTEGER
2	WAY_POINT		INTEGER
2	BEARING		DREAL
2	MOVE_POINT		INTEGER
2	PATH_POSITION_LLA		
3	LATITUDE		DREAL
3	LONGITUDE		DREAL
3	ALTITUDE		DREAL
2	PATH_POSITION_XYZ		
3	X_POS		DREAL
3	Y_POS		DREAL
3	Z_POS		DREAL
2	DISTANCE		DREAL
2	VELOCITY		DREAL
2	PATH_ROTATION		
3	X_ROT		DREAL
3	Y_ROT		DREAL
3	Z_ROT		DREAL

Figure 10-1. Resource: PATH_DATABASE.

If we want to reference the velocity of a particular plane, we could use `VELOCITY(I)`, where `I` is the pointer for a particular platform. But then we would not know what type of platform it is. To improve this, we could use `AIR_PTR` to indicate that the pointer is for an air platform. But if there are different instances of F15s and F16s, we could use `F15_PTR`, yielding `VELOCITY(F15_PTR)`, a much better designation of the platform.

Considering the resulting improvement in understandability in the code, this is not hard to do. Yet in most publications or academic examples, single character names are used over and over. Some of this is driven by people writing papers for journals that restrict page counts. So if you want to get a paper published, you must use terse representations. Another case is that of an instructor, who is pressed for time, writing examples on the blackboard in a programming class. In the later case, one can use modern facilities, e.g., showing examples from a laptop to a screen, or sharing examples on a network of computers to eliminate this restriction. As Sitner points out, this restriction can be driven by the language itself.

These considerations notwithstanding, it is clear to those experienced in overseeing major software projects that naming is critical to understanding. Furthermore, naming should be part of a hierarchical description of a complex space as illustrated in Figure 10-1.

In the literature of the 1970s it was made clear that good languages were organized to support the space, such as in Figure 10-1. TYPEing followed the structure of the space. Poor languages were organized by type, e.g., `INTEGER I, J, K`; `REAL X, Y, Z`. In addition, really poor languages allowed one to “type as you go,” i.e., one could define types as you typed - anywhere in the code. This made typing much faster. Of course it made it harder for others to find the type of a particular variable, especially when it was referenced via layers of argument lists. Years later it was even difficult for the original coder.

As described in Chapter 5, one must compare the time it takes to type code versus the time it takes another programmer to understand that code. When building complex software in a competitive environment, minimizing coding keystrokes becomes equally nonproductive. Numerous articles support this statement; see for example, [2], [66], [89], [111], [136] - [139], [141], and [146]. Considering the current popular software languages and apparent desire to produce Spartan code, this tradeoff warrants further investigation. But to get fair answers requires fair experiments, experiments that are repeatable by independent parties.

The Problem With Symbolic (Terse) Programming Languages

Some languages encourage minimization of keystrokes with their Spartan syntax and symbology. This is tantamount to minimizing information transfer and therefore understandability of the code. Such languages may reduce the time to type a line of code; but this is an extremely small part of the time spent on a module. Anyone who has had to work with another author's Spartan syntax will confirm that the time lost trying to understand, test and debug complex algorithms far exceeds that spent saving keystrokes. Our experience in working with complex software algorithms is that *terse* symbology is the wrong direction.

Symbolic languages also have the problem that loss or misplacement of a single symbol can lead to a totally different meaning, if not just a misunderstanding. An example is use of the symbols $<$, \leq , $-<$, $>$, \geq , $->$, versus spelling out the meaning, e.g., “not greater than”. Change or loss of a single symbol in the spelled out case creates an obvious visible error, clearly a case for Shannon’s theory of redundancy. This is why legal contracts require that numbers be spelled out for clarity. Changing a single (important) digit can go unnoticed, whereas changing a single character in a written number is easy to discern. That is why checks require written numbers.

One can also argue the alternative of producing documentation inside code with large blocks of comments. In fact, this is a good indicator of a poor language - one that has to be *heavily annotated* to be understood by the reader. Good languages minimize the need for comments while providing a high degree of understandability directly from the code. COBOL required very few comments, FORTRAN about 50% of a routine. It is not unusual to find well documented C-based language programs where 70% or more of the code is comments.

Grace Hopper understood this. That is why she maintained that programs should be written in a language that was close to English, and why COBOL has been the outstanding leader in the productivity race.

As determined by Jerry Sitner, [136], *Understandability* is the most important factor in human-to-human communications and the resulting productivity of teams working on large complex software systems. As told by Paul Strassmann, [141], the loner programmers do not care about this aspect of a programming language. According to Sitner, neither does the IEEE. His view was derived from the definition of the IEEE standard measurement of productivity which at the time disregarded the understandability of the language.

When a manager spends many years reading other people’s programs, trying to understand why new people cannot pick up an unfinished effort or make changes, it becomes apparent why so much time is wasted. In some cases it is wasted on simple problems. In more difficult areas, the modifications may be easy. Upon inspecting the cases that are time consuming, it is most always because it’s hard to understand what the code means - even though the algorithms may be simple. In the quick to modify cases, it is easy to understand the code, even though the algorithms are very complex. By understandability we mean: How easy is it for a new person to sufficiently understand someone else’s code so they can modify it.

There are many language factors that go into understandability. We try to cover most of them here. But it is important for the reader to understand that there are many programmers who do not agree with our measures. We have heard these reasons and have been surprised that the programmers who have stated them were that honest about their feelings. Examples follow.

When a prior employee moved to a new company and suggested they use VisiSoft, he was told that it would make their software insecure - because anyone could easily understand the algorithms. In the engineering world, particularly in the design of complex hardware systems, security of secret information is handled totally separately - by different people - from understandability of the design. Security and protection of secrets is a totally separate issue from building highly competitive designs. Why would one want to mix these issues? Are they confused about job security? What creates real job security? Detroit is a great example.

THE CASE FOR UPPER CASE

When using the English language, upper case is used typically to denote the start of a sentence or special names. These may occur in the middle of a paragraph. The additional information provided by the upper case characters is helpful in understanding the context of the words. There are no similar contexts in programming languages, unless one writes code in paragraph blocks instead of starting instructions on a new line. Differentiation between upper and lower case mostly adds confusion as explained in the next sections.

Because C-based languages are small, they depend heavily on library calls, so the libraries tend to be very large. The classic case is X-Windows, with hundreds of similar names and no hierarchy to differentiate them. This led to the use of lower and upper case to ensure differentiation without typing more than 30 to 40 characters.

When reading code, trying to find a variable that could be in upper or lower case formats is unnecessarily difficult. This is a visual scanning process. Looking for an identical pattern is much easier than looking for similar patterns. The following differences in examples of names are easy to spot since they are right next to each other, not scattered. However, if they appear in different positions in equations on different pages, they are hard to spot.

```
Critical_number = QUANTUM_NUMBER_L + 3.5e-2
Critical_Number = Quantum_Number_1 + 3.5e-2
CRITICAL_NUMBER = quantum_number_1 + 3.5e-2
critical_Number = Quantum_Number_L + 3.5e-2
```

There are two cases to be considered for upper and lower case. These are the following.

Case Insensitive - Translator Does Not Distinguish Between Upper & Lower Case

In this case, the writer can use all upper case, all lower case, or any mix desired while typing. The translator ignores the difference. All of the above statements look the same to the translator. They are obviously different to the reader when grouped as above.

When using a case insensitive language, getting agreement on standard conventions to help recognition of names is difficult. They cannot be enforced automatically. Without consistent approaches that are easily enforced they are quickly discarded. The result is that when debugging another author's code, scanning for a variable name becomes much more complex to match. One must look for all of the possible different names in the code that mean the same to the translator. Visual pattern search is very difficult in a case insensitive environment, unless the names are grouped together as in the example above. And mixing cases is incompatible with case sensitive translators.

Case Sensitive - Translator Does Distinguish Between Upper & Lower Case

In this case, the writer must carefully distinguish between upper case and lower case, since editors allow for any mix desired while typing. The translator carefully picks out the difference. In this case, all of the statements in the above example look totally different to the translator, making them incompatible with the case insensitive approach.

Upper & Lower Case Sensitivity - Driven By Large Libraries

Anyone who has used X_Windows libraries knows the difficulties encountered when typing long names correctly using upper and lower case. Worse is the difficulty in searching for problems while debugging complex graphical code. This is aggravated when referencing two or more libraries containing the same (wrong) function name, and library selection is random. Because of the size of C-based libraries, with no ability to distinguish between these libraries when linking, many names are more than 30 characters long with various mixes of case, making it hard to get them right. What's the solution? Provide a hierarchy. VisiSoft provides for calling library processes by process_name - within a specified module by module_name - within a specified library by library_name. This three-layer hierarchy greatly simplifies the selection of simple names while ensuring that one never gets the wrong library.

The Incompatibility Problem

When using Windows, C-based compilers are Case Insensitive. When using Linux or UNIX, the compilers are Case Sensitive. Moving a large piece of software from Windows (Case Insensitive) to Linux or UNIX (Case Sensitive) presents a huge incompatibility problem for software taking advantage of case insensitivity. All upper case solves all of these problems.

The Case For Fixed Width Fonts

If the argument for upper and lower case is really about style or acceptance by a popular culture, then we should consider variable width fonts. This makes reading code even more like reading text. Of course there are drawbacks. Nothing lines up, especially with numbers and decimal points

THE CASE FOR COLOR

Being able to quickly distinguish types of words (e.g., KEY words that the translator recognizes) is desirable. This may be accomplished easily using a smart editor that automatically colors KEY words; see Figures 10-2 and 10-3. More importantly, there are more choices one can use to distinguish between types of words, not just upper or lower case. As illustrated in the figure, five colors are used: black, gray, red, blue, and green.

Using this approach, one does not have to go through a translator to get the colors. The editor recognizes the KEY words and colors them automatically - *as you type*. In addition, if one mistakenly types a lower case letter that is not in a comment (started by three asterisks ***), it is automatically changed to upper case by the editor.

The VisiSoft system - all upper case - is compatible with Windows, Linux or UNIX conventions: Case Sensitive or Case Insensitive. More importantly, the built-in editor recognizes KEY words and colors them accordingly - as you type.

```

USER_FILE_CONTROLS
1  PATH_FILE_NAME          CHAR 66
1  TOTAL_PATHS             INTEGER
1  TOTAL_POINTS            INTEGER
1  POINT_ICON_POINTER      INTEGER
1  PATH_ICON_POINTER       INTEGER
1  RECS_PER_PATH           INTEGER
1  NEXT_PATH               INTEGER
1  END_POINT               INTEGER
1  PATH_PTR                INTEGER
1  PRIOR_POINT             INTEGER
1  MODIFY_PATH_NO         INTEGER

TRANSLATION_CONTROLS
1  PATH                    QUANTITY(500)
2  PATH_NO                 INTEGER
2  PATH_DATA_AREA          INDEX_1
    ALIAS FREE             VALUE 0,
    ALIAS USED             VALUE 1
2  PATH_DATA               INDEX_1
    ALIAS DELETED          VALUE 0,
    ALIAS EXISTS           VALUE 1
2  PATH_SELECTION_STATE    INDEX_1
    ALIAS NOT_SELECTED     VALUE 0,
    ALIAS SELECTED         VALUE 1
2  PATH_MODIFY_STATE       INDEX_1
    ALIAS NOT_SELECTED     VALUE 0,
    ALIAS SELECTED         VALUE 1
2  PATH_HIGHLIGHT_STATE    INDEX_1
    ALIAS NOT_HIGHLIGHTED  VALUE 0,
    ALIAS HIGHLIGHTED     VALUE 1
2  FIRST_POINT             INTEGER
2  NO_OF_POINTS            INTEGER
2  LAST_POINT              INTEGER
2  PATHNAME                CHAR 24
2  PATH_POINT_ARRAY        QUANTITY(1000)
3  PATH_POINT              ICON
3  PATH_SEGMENT            LINE
3  POINT_FINDER            INTEGER

TABLE_FILE_STATE          STATUS  CLOSED
                           END_OF_FILE
                           NOT_END_OF_FILE
                           INITIAL_VALUE  CLOSED

PATH_DATA_BASE
1  PATH_RECORD             QUANTITY(500000)
2  PATH_NUMBER_P           INTEGER
2  PATH_POINT_P            INTEGER
2  WAY_POINT_P             INTEGER
2  BEARING_P               DREAL
2  MOVE_POINT_P            INTEGER
2  LATITUDE_P              DREAL
2  LONGITUDE_P             DREAL
2  ALTITUDE_P              DREAL
2  PATH_POSITION
3  X_POS_P                 DREAL
3  Y_POS_P                 DREAL
3  Z_POS_P                 DREAL
2  DISTANCE_P              DREAL
2  VELOCITY_P              DREAL
2  PATH_ROTATION
3  X_ROT_P                 DREAL
3  Y_ROT_P                 DREAL
3  Z_ROT_P                 DREAL
2  MAX_WAY_P               DREAL
2  MAX_MOVE_P              DREAL

```

Figure 10-2. The case for COLOR to recognize types of words.

```

GLOBE_XYZ_TO_WAT_REL
  SET GLOBE_RETURN_CODE TO OK
  IF REL_ZONE_STATE IS NOT SPECIFIED
    EXECUTE FIND_ZONE_POINTERS
  ELSE
    EXECUTE TRANSFORM_COORDINATES.

FIND_ZONE_POINTERS
  MOVE GLOBE_LAT_LON_IN TO ZONE_LAT_LON
  CALL GET_ZONE_POINTERS IN COORDINATE_CONVERSIONS IN GENERAL
    USING ZONE_POINTER_INTERFACE
  IF ZONE_RETURN_CODE IS OK
    MOVE REL_ZONE_OUTPUT TO REL_ZONE
    EXECUTE TRANSFORM_COORDINATES
  ELSE
    SET GLOBE_RETURN_CODE TO INVALID_COORDINATE .

TRANSFORM_COORDINATES
  *** Find LAT-LON pointers

  THETA_11 = (LAT_POINTER_OUTPUT - 11)*8
  PHI_11   = (LON_POINTER_OUTPUT - 31)*6

  *** Get (X, Y, Z) coordinates of the (LAT, LON, ALT) point

  MOVE GLOBE_MASTER_INPUT TO ALT_DATA_INPUT
  CALL ALTITUDE_TO_XYZ IN COORDINATE_CONVERSIONS IN GENERAL
    USING LAT_LON_ALT_TO_XYZ

  IF RETURN_CODE_ALT IS NOT OK
    PRINT 'RETURN_CODE_ALT = ', RETURN_CODE_ALT .

  MOVE ALT_DATA_OUTPUT TO GLOBE_XYZ_INPUT
  EXECUTE GET_XYZ_FOR_ZONE_CORNERS
  IF LAT_POINTER_OUTPUT IS LESS THAN 11
    EXECUTE SOUTHERN_HEMISPHERE
  ELSE
    EXECUTE NORTHERN_HEMISPHERE .

GET_XYZ_FOR_ZONE_CORNERS
  *** Get the X, Y, Z points for the corners of the grid zone.

  MOVE LAT_1_LON_1 TO INPUT_DATA_LL
  CALL LAT_LON_TO_XYZ IN COORDINATE_CONVERSIONS IN GENERAL
    USING LAT_LON_XYZ_INTERFACE
  MOVE OUTPUT_DATA_XYZ TO POINT_11

  THETA_12 = THETA_11
  PHI_12   = PHI_11 + 6
  MOVE LAT_1_LON_2 TO INPUT_DATA_LL
  CALL LAT_LON_TO_XYZ IN COORDINATE_CONVERSIONS IN GENERAL
    USING LAT_LON_XYZ_INTERFACE
  MOVE OUTPUT_DATA_XYZ TO POINT_12

  THETA_21 = THETA_11 + 8
  PHI_21   = PHI_11
  MOVE LAT_2_LON_1 TO INPUT_DATA_LL
  CALL LAT_LON_TO_XYZ IN COORDINATE_CONVERSIONS IN GENERAL
    USING LAT_LON_XYZ_INTERFACE
  MOVE OUTPUT_DATA_XYZ TO POINT_21

  THETA_22 = THETA_21
  PHI_22   = PHI_12
  MOVE LAT_2_LON_2 TO INPUT_DATA_LL
  CALL LAT_LON_TO_XYZ IN COORDINATE_CONVERSIONS IN GENERAL
    USING LAT_LON_XYZ_INTERFACE
  MOVE OUTPUT_DATA_XYZ TO POINT_22

```

Figure 10-3. The case for COLOR to recognize types of words.

THE CASE FOR CONTEXT ORIENTED LANGUAGES

Context free languages are taught in compiler writing courses in Computer Science schools because the compilers for them are easy to write. Major portions can be written automatically using “Compiler Compilers.” FORTRAN is a context free language as illustrated in Figure 4-3. A basic property of context free languages is that spaces may be removed (e.g., by a PACK_LEFT routine) since scans for key words are simple.

When working with large data structures, e.g., those used in production software to access and update complex data files, hierarchical data structures are critical to run-time speed as well as ease of understanding. Moving large data structures in and out of records may be done in a single instruction fetch with all of the elementary data items being directly available. In a good language, qualifying names may be used at any higher level in the hierarchy to uniquely specify a data item below it.

This is similar to saying “pass the red book,” where red is a qualifier. This requires a sophisticated (context oriented) translator to deal with all of the complex possibilities that may occur. It puts the burden is on the translator - exactly where it should be - not on the person reading the code. See the example below.

An example of a VisiSoft hierarchical data structure is the following:

```
1  ENTRANCES
2  FRONT
   3  DOOR                STATUS OPEN
                           CLOSED
   3  WINDOW              STATUS OPEN
                           CLOSED
2  BACK
   3  DOOR                STATUS OPEN
                           CLOSED
   3  WINDOW              STATUS OPEN
                           CLOSED
```

An example of a VisiSoft conditional statement using the above data structure:

```
IF FRONT DOOR IS OPEN OR BACK DOOR IS OPEN
  EXECUTE ENTRY .
```

Visualizing Scope

The architectural approach described in the prior chapters provides a visualization of how data is shared by instructions. All data must appear in resources. *Dedicated* resources are used only by a single process, and have only a single connect line to that process. *Shared* resources are connected to those processes which share that data.

Forcing every data element into this structure is a great simplification. It ensures immediate visibility of what instructions have access to what data. It forces designers to put more thought into architectures that promote independence, therefore affording ease of restructuring. More importantly, one can determine from a quick visual inspection of an engineering drawing which processes share a resource, and therefore any subset of a data structure.

An architect can minimize the connectivity between processes at the drawing level. Since it is the connectivity that determines independence, this serves to maximize independence of modules, supporting software designs that are best suited to parallel processing.

Many organizations impose coding restrictions above and beyond those in a given language to attempt to achieve some level of independence. However, without preprocessors, and with access to individual variables as in C-based languages, one must track down all the functions that contain data definitions of each individual variable used by a function to enforce a standard, or to locate a problem in that function. Languages with the potential for creating high connectivity make it difficult to achieve independence.

Separation of data from instructions in the development environment produces an entirely new view of scope, one that is specified precisely in the architectural drawing. Since there is a direct mapping from architecture to code, language design in this paradigm is not concerned with scope. More importantly, allowing scope changes in the language conflicts directly with architectural control. Since architecture is at a higher level of purview in the design chain, and provides direct visualization of independence, scope is best determined at the architectural level.

Simplifying Subroutines

In the CAD system described here, a *process* contains the instructions that act on data. At first glance, a process may look like a *function* or *subroutine* in a typical programming language. However, many of the problems we deal with today are dominated by sets of rules that operate on complex data structures, not just mathematical functions. This implies many large dissimilar transformations that must be tailored to the complex data structures upon which they operate.

In current popular programming languages, the lack of strict one-in one-out control structures (as described by Mills, [102]) creates problems in the support phase of a software product. Using a C-based language, the resulting code can contain complex conditional statements implemented with many layers of nested brackets. This becomes a problem when new features are added to a function and the number of conditions must grow to support these features. Unless one takes the time to break up the C function into multiple subroutines - or resorts to the use of GOTOs - the nesting continues to increase.

Since breaking up a complex nested conditional statement is difficult, and creating a new function (subroutine) means deciding what data must be available to the new function, the nesting typically grows. Using C, the local data becomes global. If C⁺⁺ is used, it can be shared among “friends.” These decisions are all practical deterrents to breaking up the nests. And as they grow, they become more difficult to maintain.

So what is the solution? Add another layer of hierarchy into the process (hierarchies serve to control complex organizations). This is implemented by inserting *rules* into a process, whereby a rule may be executed as a one-in one-out control structure, including the ability to execute a rule multiple times. Examples of this improvement are described in subsequent sections.

Sharing Data – Elimination of Argument Lists

Most older programming languages use argument lists when calling subroutines or functions. A simple example is SIN(THETA). This approach is handy for built-in scientific and Boolean functions. However, when passing multiple arguments, things can easily get confused. Unless otherwise specified, the data values in the argument list are copied to new memory locations before being operated upon by the function. Upon return from the function call, they may or may not be copied back into memory locations associated with the calling routine.

When nonnumeric data structures are used, passing data becomes cumbersome and error-prone. For medium size data structures, processing time can become troublesome. In these cases, many languages provide for passing pointers to the data, instead of passing the data. However, it is up to the programmer to manage and pass the pointers. Using the CAD approach defined here, resources are shared directly by those processes to which they are connected in the architecture. Access is always by pointers that are managed automatically behind the scenes.

If the designer wants to make a copy of an attribute structure, he simply copies that structure to a corresponding structure, typically in a different resource. The original attribute structure remains in tact. Often one may want to look at the data using a different structure in the receiving resource. This can only be accomplished if the attribute structures preserve their size and structure when moved. Because of the well specified nature of hierarchical data structures in a resource (What You See Is What You Get), and the corresponding group move property, this is simple to do.

CHAPTER 11 *MEMORY RESOURCE DESCRIPTIONS*

11.1 **DECLARATION OF DATA**

The Problem of Global Types

If one tries to use some of the more complex C-code functions, e.g., those for graphics or Inter-Process Communications (IPC), one typically runs into the problem of variable data type definitions, or *defined data types*. These types are defined in one C routine, and used in another. In C⁺⁺, the type can be in a “class” that is shared.

The problem encountered with user defined data types occurs when trying to locate the data type definition in the routine of interest, and determining that it is defined elsewhere. One must then locate the source code for the defining routine to determine the data type. Having found the source code for the defining routine, and then the data type definition, one may discover that this definition depends upon yet another defined data type in yet another routine. When one finally gets to the end of such a chain, one should not be surprised to discover that the data type is simply an integer. Such mystery chains are not uncommon in large systems when user defined data type definitions are permitted, and people elect to *redefine the dictionary*.

When data declarations are imbedded in code with instructions, programmers are inclined to minimize the size of data declarations, or just minimize the keystrokes. This is most evident when reading C-based code, where programmers use terse data definitions that allow them to *type* the data as they go, within arithmetic or conditional constructs, making it hard to identify the type definition. Such practices clearly minimize understandability along with keystrokes.

Clarity is enhanced with a *single fixed dictionary* for all data types. Enforcement of this principle becomes simple if data typing is not mixed with the executable statements, but placed in a separate entity. Since resources are separate entities that are defined and shared by processes that want access to them, then by virtue of connect lines in the architecture, there are no data declarations in a process. Therefore, the *re-typing* problem does not exist.

The presence of global data types generally creates a dependency with any subprogram that shares them. This lack of independence creates severe bottlenecks for parallel processing. With the CAD approach described above, the architect determines explicitly - by design - what processes have access to what resources. The connectivity is clear, visually, from the drawing. *There are no global types.*

Machine Independent Standards (What You See Is What You Get!)

Older languages, e.g., C, C⁺⁺ and their derivatives, perform *word boundary alignment* of arithmetic type data. Yet data has not been organized as words since the birth of the “byte” in the IBM 360 (1960s). Word boundary alignment causes data structures to get *padded* with “slack” bytes, implying that group level structures are not stored as defined by the programmer’s code. Moving group level structures will generally cause data to be inserted incorrectly into another structure, unless great care is taken to define the structures on precise word boundaries. This is a very undesirable restriction when working with databases or performing message or symbol string processing. Even when compilers offer an option to ignore word boundary alignment, programmers ignore the option for fear of incompatibilities across a team.

Using Hierarchies To Control Complexity

As stated in the introduction, software must deal with ever increasing levels of complexity. Complex organizations are best understood and controlled using hierarchies. The number of hierarchical levels must be sufficient to push down the complexity. This makes a system easy to understand. Languages can be designed with hierarchical properties that aid in visualization of the organization, clearly simplifying understandability.

Hierarchical structures are a critical property of software languages. This principle was clearly understood by the world's best language designer, Grace Hopper. Hierarchies are a major factor in ease of understanding. They simplify the specification of complex data spaces as shown in Figure 11-1. In turn, data spaces such as these can support great simplification of complex algorithms. As described further in this chapter, hierarchies also apply directly to the simplification of complex instruction sets. The proper use of hierarchies in a software language requires data structures that are easily created, referenced, and understood. Equally important, the structures must be organized and grouped to match the application - *not grouped by type*.

RESOURCE NAME: MESSAGE_FORMATS		INSTANCES: TRANSMITTER RECEIVER
MESSAGE		
1	SYNC_CODE	CHAR 6
	ALIAS VALID	VALUE '101010', '010101'
1	TYPE	STATUS FORMAT_A FORMAT_B
1	CONTENT	CHAR 46
FORMAT_A REDEFINES MESSAGE		
1	PAD	CHAR 14
1	HEADER	
	2 PRIORITY	STATUS FLASH IMMEDIATE ROUTINE
	2 ORIGIN	INDEX
	2 DESTINATION	INDEX
	ALIAS BROADCAST	VALUE 0
1	BODY	
	2 LENGTH	INTEGER
1	TRAILER	
	2 MESSAGE_NUMBER	INTEGER
	2 TIME_SENT	REAL
	2 TIME_RECEIVED	REAL
	2 ACKNOWLEDGEMENT	STATUS RECEIVED NOT_RECEIVED
	2 LAST_SYMBOL	CHAR 2
	ALIAS TERMINATOR	VALUE '\\', '/ /', '<<', '>>'
FORMAT_B REDEFINES MESSAGE		
1	PAD	CHAR 14
1	HEADER	
	2 SOURCE	INDEX
	2 SINK	INDEX
1	BODY	
	2 CONTENTS	CHAR 42

Figure 11-1. Example of hierarchical data structures in a resource.

Resource structures are byte oriented and fully compatible with current and planned chip designs. Since data on today's chips is byte addressable, a designer can create complex hierarchical structures that are most convenient for meeting requirements. What You See (in your attribute structure) Is What You Get (in memory), independent of the machine you are using. Clearly the language translators are much more complex since the burden is shifted from the developer to the computer - at translation time. However, the real achievements occur at run-time with dramatic increases in speed.

Achieving Speed With Hierarchical Group Moves

The ability to move a complete hierarchical structure, or any substructure within a hierarchy, with a simple MOVE statement is important to the implementation of transformations in complex systems. This permits group moves of one complex structure or substructure to another with a single instruction. Not only does this simplify the code, it requires only a single instruction fetch, dramatically reducing running times. A good example is moving complete messages, or fields containing subfields, as illustrated in Figure 11-1. These speed differences are demonstrated in Chapter 17. When creating communications protocols or simply transferring records on a file, one may want to move subfields into a packet, and packets into a frame. These group moves are executed simply by referring to the attribute name of the highest level group to be moved. One need not worry about its size or structure. The implementer need only ensure that the receiving structure is organized to properly receive the data being moved. This reduces instruction code as well as running time.

The ability to use *hierarchical group moves* results from the *machine independence* properties described above. Hierarchies are easy to define and easy to understand using the approach shown in Figure 11-1. Ease of use in a process is also significantly improved by minimizing the qualification of names in a hierarchy to that sufficient for unique identification.

Reusable Names And Qualifiers

It is common to have many thousands of attributes in a large system, and the selection of names in large modules becomes a critical part of helping others to understand the module. In the CAD environment described here, names may be reused without qualification except in the same process. Within a single process, one does not have to make up different names to distinguish the same type of data from another when they are obviously used in different contexts.

For example, when reading the specifications for two different equipment locations, one would expect to find the same words reused, possibly with other names as qualifiers, e.g., RADIO LOCATION or ANTENNA LOCATION. One can easily understand the meanings in accordance with the context of each separate specification.

VisiSoft translators recognize the context of attribute names and qualifiers. This frees the user from having to create needlessly different names that mean the same thing in different contexts. What's more, this is done without injecting special delimiters, e.g. an underscore or period, to signify qualification. This implies translation based upon context; again, putting a heavy burden on the translator - not the user.

11.2 LEVEL NUMBERS

Level numbers are used to organize a resource into a hierarchical structure of group and elementary attributes. Level numbers subdivide a resource, and once a subdivision has been specified, it may be further subdivided to permit more detailed resource reference.

The basic subdivisions of a resource, that is, those not further subdivided, are called *elementary attributes*; consequently, a resource will consist of one or more elementary attributes.

In order to refer to a set of elementary attributes, the attributes are combined into groups. A *group attribute* consists of a named collection of one or more elementary attributes. Group attributes, in turn, may be combined into larger groups. Thus, an elementary attribute may belong to a set of nested groups. A maximum of 800 lines are allowed within each unnumbered group attribute.

A system of level numbers shows the hierarchical structure of elementary attributes and group attributes. The first or "top-level" attribute cannot have a level number. For each level of subdivision, level numbers must be assigned in increasing order. Except for the top level group, the integers 1, 2, 3, ... are used as level numbers. Level numbers are integer value less than or equal to 45 and must be contiguous starting from the top.

Other than top level attributes starting in columns 1 to 4, and all others in 5 or beyond, no spacing rules are enforced in GSS. However, the benefits of following indentation and alignment conventions for improved readability are obvious.

Figure 11-1 illustrated the hierarchical nature of the VisiSoft Resource, and the level of complexity of a typical shared resource. Internally, each resource statement has the following format:

```
[level_number]  attribute_name  [data type]  [qualifying clauses]
```

Each statement begins on a new line and the different parts of the statement are separated by one or more spaces. Top-level group attribute statements (see below) must begin in columns 1 through 4. All other resource statements must begin in column 5 or beyond. A maximum of 72 characters are allowed per line. If necessary, statements may be continued on to more than one line, although qualifying clauses may not be split between lines (see below).

Example

```
INPUT_MESSAGE
  1  ORIGIN                      INTEGER
  1  DESTINATION                 INTEGER
  1  MESSAGE
    2  MESSAGE_HEADER            CHAR 10
    2  MESSAGE_BODY              CHAR 24
    2  MESSAGE_TRAILER           CHAR 10
  1  ARRIVAL_TIME                INTEGER
  1  DURATION                    INTEGER
  1  TYPE                        CHARACTER 5
OUTPUT_MESSAGE                   CHARACTER 50
```

11.3 ATTRIBUTE NAMES

Attribute names must be GSS words (limited to 32 characters, starting with an alphabetic character). When used in a resource, attributes at the same level (directly subordinate to the same attribute) must have unique names. Note: Since top-level attributes are directly subordinate to a resource, they must have unique names.

Attribute names used in a process must be uniquely qualified. This can be accomplished by using any combination of higher level names in the hierarchy. Note: Resource names are at the top of the hierarchy.

REUSE OF ATTRIBUTE NAMES

Reuse of names that refer to different physical attributes in a GSS process is allowed provided the intended use could be uniquely resolved. For example, the use of the name DOOR to mean FRONT DOOR or BACK DOOR is resolved by adding the qualifier FRONT or BACK. Consider the following example from a GSS resource description.

```
1 ENTRANCES
2 FRONT
3 DOOR          STATUS      OPEN
                  CLOSED
3 WINDOW        STATUS      OPEN
                  CLOSED
2 BACK
3 DOOR          STATUS      OPEN
                  CLOSED
3 WINDOW        STATUS      OPEN
                  CLOSED
```

A GSS conditional statement using that resource can be written as follows:

```
IF FRONT DOOR IS OPEN OR BACK DOOR IS OPEN
    EXECUTE ENTRY
ELSE IF FRONT WINDOW IS OPEN OR BACK WINDOW IS OPEN
    EXECUTE CHECK_ENTRY
ELSE ...
```

Although the same names (e.g., DOOR) are reused in the resource, the intended use is resolved by using the qualifiers FRONT or BACK in the process. In the case of OPEN or CLOSED, reuse of STATUS names is qualified automatically by the particular status attribute FRONT DOOR or BACK DOOR. This is also true for alias names. They are automatically qualified by the attribute names that use them.

Names are reusable within the same resource or over multiple resources. Reuse of names in a GSS process requires qualification only to the extent sufficient to insure uniqueness at the lowest (innermost) level in the resource hierarchy. For example:

```

HIGH_POWER
  1  TRANSCIEVER
    2  HEIGHT          REAL
    2  LOCATION
      3  X_LOCATION    REAL
      3  Y_LOCATION    REAL

LOW_POWER
  1  TRANSCIEVER
    2  HEIGHT          REAL
    2  LOCATION
      3  X_LOCATION    REAL
      3  Y_LOCATION    REAL

```

One can write the following statements:

```

MOVE HIGH_POWER HEIGHT TO LOW_POWER HEIGHT
MOVE HIGH_POWER X_LOCATION TO LOW_POWER X_LOCATION

```

This is also true for ALIAS names as defined below. They are automatically qualified by the attribute names that use them.

MULTIPLE INSTANCED ATTRIBUTE STRUCTURES

Description of multiple instances of attribute structures is implemented using the QUANTITY clause in GSS. For example an incoming message buffer may provide for up to 20 stored messages at any instance of time. This requires 20 slots in the buffer, each slot being able to store one message.

At any instance of time, we may want to check certain fields, e.g., MESSAGE_PRIORITY, in all of these messages to determine which message to pull from the input buffer next. The resource structure to support this may appear as follows:

```

MESSAGE_BUFFER
  1  MESSAGE          QUANTITY(20)
    2  MESSAGE_HEADER
      3  MESSAGE_TYPE    CHAR 8
      3  MESSAGE_PRIORITY STATUS LOW
                                MEDIUM
                                HIGH
    2  MESSAGE_BODY    CHAR 68

```

Each MESSAGE (there are 20) has a MESSAGE_HEADER and MESSAGE_BODY, each storing its own MESSAGE_TYPE and MESSAGE_PRIORITY by virtue of the QUANTITY clause.

11.4 DATA TYPES AND QUALIFYING CLAUSES

Each elementary attribute must have a data type clause to describe the values it may assume, and may also have a qualifying clause. Group attributes may either have no type or qualifying clauses, or one QUANTITY clause as described below. The order of clauses within a statement is not important. The possible *type* clauses are defined below within groups of types:

```
NUMERIC
    INTEGER
    INDEX
    INDEX_1
    REAL
    DREAL
    COMPLEX
    DCOMPLEX

DECIMAL
    DECIMAL

CHARACTER
    CHARACTER
    STATUS
    COLOR

CONTROL
    RULE
    PROCESS
    EVENT
```

For character data, the number of bytes used is specified using the CHARACTER or DECIMAL data type clause. For numeric type data, the specified types are given in the following sections. For status attributes, the number of bytes used is equal to the length of the longest status name. Additional *qualifying* clauses are as follows:

```
QUANTITY
ALIAS
REDEFINES
INITIAL_VALUE
```

All of the above are described in the sections that follow.

11.4.1 NUMERIC DATA TYPES

Numeric attributes have special properties because of the way they are stored. This is to allow for fast binary computations. These types are described in the following sections.

11.4.1.1 INTEGER Attributes

The INTEGER clause specifies that the attribute is expected to be any positive or negative integer. An INTEGER attribute may assume positive or negative values with 9 digits of precision in the range -2,147,483,648 to +2,147,483,647. Integer takes 4 bytes of storage.

11.4.1.2 INDEX Attributes

An INDEX attribute specifies an integer that may assume positive or negative values with over 4 digits of accuracy in the range -32,768 to +32,767. The INDEX attribute is stored as a binary number and occupies 2 bytes of storage. The abbreviation INDX is recognized by GSS as equivalent to the keyword INDEX.

11.4.1.3 INDEX_1 Attributes

The INDEX_1 attribute specifies an integer that may assume positive or negative values with over 2 digits of accuracy in the decimal range [-128, +127]. The INDEX_1 attribute is stored as a binary number and occupies 1 byte of storage.

INTEGER, INDEX, and INDEX_1 examples

FLIGHT_CONTROLS	
1 NUM_CHANGES	INTEGER
1 MISSION_TIME	INTEGER
1 AIRCRAFT_TRANS_NUM	INDEX
1 CHANGE_NUM	INDEX
1 TRANSACTION_TYPE	INDEX_1
1 CHANNEL_NUMBER	INDEX_1

11.4.1.4 REAL and DREAL Attributes

The REAL and DREAL clauses specify that the attribute is expected to be a positive or negative real or double precision real number. A REAL attribute is represented by a floating-point real number in the format: $\pm.999999E\pm99$

The mantissa (portion before the 'E') is a signed decimal number with 6 digits of precision. The exponent (portion after the 'E') is a signed two-digit integer whose value must lie in the range [-38, +37]. It occupies 4 bytes of storage.

A double precision real attribute is specified by the reserved word DREAL. The mantissa of a double-precision number is a signed decimal number with 15 digits of precision. The exponent must lie in the range [-308, +307]. It occupies 8 bytes of storage. A DREAL attribute is represented by a floating-point real number in the format: $\pm.999999999999999E\pm999$

REAL and DREAL examples

TRANSCIEVER_LOCATION	
1 X_LOCATION_TR	REAL
1 Y_LOCATION_TR	REAL
1 ELEVATION	DREAL
ANTENNA_GAIN	REAL
SAVED_CLOCK_VALUE	DREAL

11.4.1.5 COMPLEX And DCOMPLEX Attributes

The COMPLEX clause specifies that the attribute is a complex number $x + iy$, where i is the imaginary part. It is represented by an ordered pair, (x, y) , of REAL numbers. It occupies 8 bytes of storage. Refer to the above section for the definition of a REAL number.

A double precision, complex number may be specified by using the reserved word DCOMPLEX in place of COMPLEX. It is represented by an ordered pair, (x, y) , of DREAL numbers. It occupies 16 bytes of storage. Refer to the above section for the definition of a DREAL number.

Both COMPLEX and DCOMPLEX numbers can be redefined as a pair of REAL and DREAL numbers respectively.

COMPLEX and DCOMPLEX examples

HIGH_VOLTAGE_VALUES	
1 LINE_1	COMPLEX
1 LINE_2	COMPLEX
TOTAL_VOLTAGE_VALUES	DCOMPLEX

11.4.2 DECIMAL DATA TYPES

The DECIMAL clause is used for formatting attributes to be displayed, printed, or written to external files. DECIMAL is a character field; therefore no *arithmetic* may be performed on attributes of type DECIMAL. The optional suppression qualifying clauses are useful for reporting on real-valued attributes, without restrictions to a specified floating-point format of a REAL or DREAL attribute (16 digit mantissa followed by a 3 digit exponent).

The symbol_string following the keyword DECIMAL (or DEC) specifies how the attribute is to be represented. The symbol_string may be composed of the following symbols:

- + Indicates that a character position will be reserved to print the sign of the number *whether it is positive or negative*. The sign occupies one byte of storage.
- Indicates that a character position will be reserved to print the sign of the number *only when it is negative*. The sign occupies one byte of storage.
- .
- E+ Indicates an exponent will appear explicitly in printed output. E+ occupies two bytes of storage. E+ must be placed between the mantissa - *which must start with a decimal point* - and decimal digits representing the exponent.
- 9 Indicates that a character position will contain one of the digits 0-9. One byte of storage per digit is occupied.
- () Parentheses are used to indicate the number of occurrences of the 9 symbol. For example, 9(3) is equivalent to 999 and will occupy three bytes of storage.

The above decimal qualifying symbols produce numbers (*no internal spaces*) that can be read as character data on input using a matching DECIMAL symbol_string or CHARACTER field. They can also be converted into binary number fields using the CONVERT statement.

DECIMAL Examples (NOTE: b implies a blank space.)

SENDING_FIELD CONTENT	RECEIVING_FIELD FORMAT	OUTPUT_RESULT
+11.5	DECIMAL 999.99	011.50
+1.5	DECIMAL -9(2).9(3)	b01.500
+1295.5	DECIMAL +.9(3)	+.500
+95.	DECIMAL +.99E+9(2)	+.95E+02
+.0095	DECIMAL +.99E+9(2)	+.95E-02
-1.	DECIMAL +.9(9)E+9(3)	-.100000000E+001

Values may only be assigned explicitly to DECIMAL attributes by using a MOVE statement (see Section 9.1.3). Numeric assignment statements (Section 9.1.1) are not valid. Note that truncation will occur when a value MOVED into a DECIMAL attribute is too large.

Decimal Attribute Output Display Formatting

Additional symbols are available for formatting output displays. *Attributes using these symbols are not allowed in sending fields in a MOVE statement.* They are designed to create spaces or shift digit positions to accommodate readable report formatting

- Z Indicates zero suppress (replace zeros by spaces) under a MOVE statement, refer to Section 9.1.3. Z's may not appear following a 9, L, R or the explicit decimal point, or in a symbol_string containing the exponent E+.
- L Indicates left justification of all symbols after zero suppress. L's may not appear following a 9, Z, R, or the explicit decimal point, or in a symbol_string containing the exponent E+.
- R Indicates right justification of the sign field prior to leading zeros after zeros are suppressed. R's may not appear following a 9, Z, or L, or the explicit decimal point, or in a symbol_string containing the exponent E+.
- () Parentheses are used to indicate the number of occurrences of a symbol. For example, 9(3) is equivalent to 999 and will occupy three bytes of storage. L(3) is equivalent to LLL and will occupy three bytes of storage.

Display Formatting Examples (NOTE: **b** implies a blank space.)

SENDING_FIELD CONTENT	RECEIVING_FIELD FORMAT	OUTPUT_RESULT
+11.5	DECIMAL Z(2)9.99	b 11.50
+1.5	DECIMAL +Z(2)9.99	+ bb 1.50
+1295.5	DECIMAL +L(2)9(2)	+1295
+95.	DECIMAL -L(2)9(2)	b 95 bb
-1.	DECIMAL L(3)9(2)	01 bbb
+11.5	DECIMAL +9(4).9(2)	+0011.50
+11.5	DECIMAL +Z(2)9(2).9(2)	+ bb 11.50
+11.5	DECIMAL +R(2)9(2).9(2)	bb +11.50
+11.5	DECIMAL +R9(3).9(2)	b +011.50

When using a DECIMAL attribute in a class or relation conditional construct, it must be considered character data after suppression or justification is applied.

11.4.3 CHARACTER DATA TYPES

11.4.3.1 CHARACTER Attributes

The CHARACTER clause specifies that the attribute is expected to take on alphanumeric values, that is any nonnumeric literal value with one byte per character, followed by an integer. The integer represents the length of the character string, and may take on values from 1 to 999999. If no integer is specified, the default is CHARACTER 1.

CHARACTER examples

```
INPUT_MESSAGE
  1 MESSAGE_HEADER      CHARACTER 10
  1 MESSAGE_BODY        CHAR 54
```

11.4.3.2 STATUS Attributes

The STATUS clause is used to indicate each of the allowed states which an attribute may assume during a simulation run. Each state is identified by a *status name*. The length of the longest status name (number of characters) determines the size of storage for that STATUS attribute. One byte of storage per character is occupied. The list of status names may appear on one line, separated by commas or on separate lines.

Up to 50 status names may be used for STATUS attributes. Within a resource, the same status name may be used more than once provided it is unique within each STATUS attribute.

STATUS examples

```
TRANSCIVER          STATUS TRANSMITTING ,  
                     RECEIVING  
PROBABILITY          STATUS LOW, MEDIUM, HIGH
```

STATUS attributes can only take on the specified values.

11.4.3.3 COLOR Attributes

The COLOR attribute allows one to assign colors from the built-in color map in RTG. Refer to the RTG manual for the selectable color names and shade numbers. The size of a color attribute is 24. Shades can be specified by adding the shade number (i), [i = 1, 2, ..., 64], e.g., RED(24). If shade is not specified, it is set to 32, the default color. A total of 2560 colors and shades are available for foreground and *background* colors and ramps. Of these, 2016 colors are predefined using the color names. The remaining 545 can be redefined by the user for color ramps.

COLOR examples

```
PROBABILITY          COLOR  
CONNECT_LINE         COLOR GREEN,  
                     YELLOW( 2 ) ,  
                     RED( 28 )  
CABLE_TYPE           COLOR INITIAL_VALUE YELLOW( 2 )
```

11.4.3.4 RULE Attributes

The RULE clause is used to define each of the allowed rule names that a rule_pointer attribute can assume during a simulation run. It is used to support the RULE_POINTER version of the CASE statement as described in Section 9.2.5 as well as conditional statements. An attribute with a RULE clause is called a rule-pointer attribute.

The list of rule_names may appear on one line, separated by blanks or commas or on separate lines. Rule_names must be GSS words (section 7.6.4) and must always be in column 5 or beyond. Each rule_name must correspond to a rule name in the process that contains the single construct CASE statement invoking the RULE clause. The length of the longest rule name (number of characters) determines the size of storage for that RULE attribute. One byte of storage per character is occupied.

There may be up to 50 rule_names for each RULE attribute. Within a resource, the same rule_name may be used more than once; however it must be unique within each rule_pointer attribute. There may be a maximum of 50 rule-pointer attributes within one resource.

RULE examples

```
RULE_POINTER_NAME    RULE INITIALIZE_NETWORK  
                     ROUTE_NEXT_CALL  
                     DISCONNECT_CALL  
  
CURRENT_SECTION      RULE CONTROL_SECTION  
                     PROCESS_SECTION
```

11.4.3.5 PROCESS Attributes

The PROCESS pointer clause is used to define each of the allowed process names that a PROCESS pointer attribute can assume during a simulation run. It is used to support the PROCESS pointer version of the CALL statement as described in Section 9.2.6 as well as conditional statements. An attribute with a PROCESS pointer clause is called a PROCESS pointer attribute.

The list of process_names may appear on one line, separated by blanks or commas, or on separate lines. Process_names must be GSS words and must always be in column 5 or beyond. Each process_name must correspond to a process in the task that contains the process invoking the PROCESS pointer clause. The size of storage for the PROCESS pointer attribute is always 4 bytes, since the PROCESS pointer is stored as an INTEGER, and is thus a numeric attribute.

There may be up to 50 process-names for each PROCESS pointer attribute. Within a resource, the same process_name may be used more than once; however it must be unique within each PROCESS pointer attribute. There may be a maximum of 50 PROCESS pointer attributes within one resource.

PROCESS Name examples

```
PROCESS_POINTER_NAME  PROCESS  INITIALIZE_NETWORK
                           ROUTE_NEXT_CALL
                           DISCONNECT_CALL

NEXT_PROCESS          PROCESS  CONTROL_TIMERS
                           DRAW_TERRAIN
                           COMPUTE_MEASURES
```

11.4.3.6 EVENT Attributes

EVENTs may occur at any level in a computer system, and must be handled by the OS. EVENT attributes support events occurring in separate tasks or within a single task running on different processors. To be recognized, EVENTs must be defined in those resources, within different tasks or IND Modules, that are shared by the processes that use them. The data type is EVENT and stored as an INTEGER. It may only take on the values of 0 or 1.

An example of the use of the EVENT attribute along with the ALIAS is shown below:

```
1  GATE_SIGNAL_5          EVENT
                           ALIAS GO          VALUE 1
                           ALIAS STOP        VALUE 0
```

We note GO could be aliased to VALUE 0 and STOP could be aliased to VALUE 1. The allowed values are binary but the assigned ALIASes can be anything that the user wants to read in an event condition to determine whether to wait or proceed. This is a matter of using redundancy to ensure understandability.

11.5 QUALIFYING CLAUSES

This section defines the qualifying clauses used in VisiSoft.

11.5.1 QUANTITY Clause

The QUANTITY clause is used to create vector, tabular or generally recurring attribute structures. It eliminates the need for separate entries for recurring attributes since it indicates the number of times an attribute or set of attributes with identical structures is repeated. It also supplies information required for the application of subscripts.

This format is used to specify a fixed-sized table. The integer specified within parentheses represents the exact number of occurrences, and must be greater than zero. It must appear on the same line as the QUANTITY keyword in the range [1, 999999].

The other resource clauses associated with an entry whose attribute includes a QUANTITY clause apply to each occurrence of the attribute described. In particular, a group of attributes may be repeated by specifying a QUANTITY clause for the group attribute. A maximum of six nested levels of the QUANTITY clause is allowed.

The attribute names of the entries that contain the QUANTITY clause, and any entries within it, must be subscripted whenever they are referred to in any statement of a process. Refer to Section 7.6.5. The number of subscripts must match the number of hierarchical levels of quantity clauses that affect the attribute. The order of the subscripts must match the order of the hierarchy, from top to bottom.

To refer to a group of recurring attributes, for example when assigning identical values (such as zero or spaces) throughout the group, they must be specified using the group name. See the group MESSAGE_GROUP in the examples below.

Note that when initializing or moving values at a group level, unpredictable results may occur if the attributes within a group have different qualifying clauses. To set zero values for INTEGER, INDEX, INDEX_1, REAL, and DREAL each individual attribute must be separately initialized with a MOVE ZERO statement. To initialize a group of INTEGER, INDEX, and INDEX_1 attributes, use a MOVE LOW_VALUES statement at the group level. The MOVE LOW_VALUES statement cannot be used to initialize REAL and DREAL attributes in a group. The abbreviation QUAN is recognized by GSS as equivalent to the keyword QUANTITY.

QUANTITY examples

```
MESSAGE_GROUP  QUANTITY(50)
  1  INPUT_MESSAGE                                CHARACTER 20

TRANSCIVER_CONNECTIVITY  QUANTITY(500)
  1  TRANSCIVER_CONN_RC  QUANTITY(500)
    2  ATTENUATION_FACTOR                                REAL
    2  SIGNAL_POWER                                     REAL

RECEIVER  QUANTITY(500)
  1  RECEIVER_CONN                                REAL
  1  RECEIVER_PWR                                 REAL
```


QUANTITY Clause – Creating Complex Tables

The quantity clause allows the modeler to define multi-dimensional tables, which themselves are hierarchical. This is a valuable property of the resource structure. In the example shown above, the links between 500 transceivers can be evaluated by checking the link between every receiver (there are 500) and all other possible transmitters (there are 500 - 1).

```
IF LINK(RECEIVER, TRANSMITTER) IS POOR
    SET TRANSCIEVER_RULE TO TURN_OFF_RECEIVER
ELSE
    IF LINK(RECEIVER, TRANSMITTER)) IS GOOD
        SET TRANSCIEVER_RULE TO RECEPTION .
```

11.5.2 ALIAS Clause

The ALIAS clause enables a group of values to be identified by a single collective identifier called the *alias name*. The alias_name is a GSS word, and must follow the rules given in Section 11.3. Reserved words may be used for alias names. It is used to check broad conditions (alias conditions) as described in Section 12.2.1.6. The ALIAS clause may be used along with a CHARACTER, DECIMAL, INTEGER, INDEX, INDEX_1, REAL, or DREAL clause for an elementary attribute. It may not be used alone or with a STATUS clause.

The ALIAS clause must immediately follow the other qualifying clause associated with the attribute. The list of numeric or nonnumeric literals, separated by commas, specify the group of values which are to be associated with the alias name. The type of literals specified must be consistent with the other clause for that attribute in terms of class (numeric or nonnumeric) and length. Each literal may be up to 24 characters long, and there may be up to 50 literals for one alias_name. More than one ALIAS clause may be specified for one attribute. The ALIAS clause may be applied to a group attribute.

ALIAS examples

INPUT_MESSAGE	
1 LEAD_CHARACTER	CHAR 1
ALIAS CONTROL_CHAR	VALUE 'S', 'R'
ALIAS DELIMITER	VALUE '.', ',', ';', ':'
1 MESSAGE_TEXT	CHAR 78
1 LAST_DIGIT	INDEX_1
ALIAS TERMINATOR	VALUE 0,9

Defining Properties That Can Be Tested Easily

When large quantities of conditional statements are nested to specify what action must be taken, clarity is lost. One must be able to represent compound conditions easily. One also wants to read and understand the conditions quickly and reliably. This requires potential states of attributes to be represented by names of groups or sets of states.

Consider,

```
CALENDAR_INFORMATION
1  MONTH                                CHARACTER 9
    ALIAS THIRTY_DAY_MONTH              VALUE 'APRIL      ',
                                           'JUNE        ',
                                           'SEPTEMBER',
                                           'NOVEMBER  '

    ALIAS THIRTY_ONE_DAY_MONTH           VALUE 'JANUARY   ',
                                           'MARCH      ',
                                           'MAY        ',
                                           'JULY       ',
                                           'AUGUST     ',
                                           'OCTOBER    ',
                                           'DECEMBER   '

1  YEAR                                STATUS NORMAL,
                                           LEAP_YEAR
```

The attribute structure above supports the following conditional statement:

```
IF MONTH IS A THIRTY_DAY_MONTH
    EXECUTE THIRTY_DAY_RULE
ELSE IF MONTH IS A THIRTY_ONE_DAY_MONTH
    EXECUTE THIRTY_ONE_DAY_RULE
ELSE IF YEAR IS A LEAP_YEAR
    EXECUTE TWENTY_NINE_DAY_RULE
ELSE EXECUTE TWENTY_EIGHT_DAY_RULE.
```

In this example, the ALIAS qualifier names a set of VALUES that MONTH can take on. The names of a month (in quotes) are used as the VALUES of the attribute to be tested or printed directly. The STATUS attribute type provides discrete character states that can be SET as well as tested directly. The reader of this conditional statement does not have to refer to any other documentation to understand the conditions for setting and testing these attributes.

11.5.3 REDEFINES Clause

The redefines clause may be used along with one of the elementary attribute qualifying clauses (INTEGER, INDEX, INDEX_1, DECIMAL, REAL, DREAL, or CHARACTER) or to specify a group attribute. Its purpose is to redefine a data structure into a new structure, typically at a group level. Any attributes linked by a REDEFINES clause must have the same level number and must be the same size. REDEFINES clauses may be nested.

REDEFINES examples

```
INPUT_MESSAGE                                CHARACTER 30

MESSAGE_DETAIL REDEFINES INPUT_MESSAGE
  1 MESSAGE_SENDER                           CHAR 10
  1 TIME_SENT                                INTR
  1 AMOUNT_WANTED                             REAL
  1 MESSAGE_RECEIVED                          CHAR 12

PARTS_NEEDED    QUANTITY(10)
  1 PART_NUMBER                                DECIMAL 9(4)
  1 DETAIL_CODE    REDEFINES PART_NUMBER
    2 DEPT_NUMBER                            CHAR 2
    2 ITEM_NUMBER_1                          CHAR
    2 ITEM_NUMBER_2                          CHAR
  1 NUMERIC_CODE    REDEFINES DETAIL_CODE
    2 .                                        INDEX
    2 .                                        INDEX_1
    2 .                                        INDEX_1
```

11.5.4 INITIAL VALUE Clause

The INITIAL_VALUE clause allows values to be assigned to resource attributes at the beginning of a simulation run. It may be used alone (for a group attribute) or following any of the other qualifying clauses. For numeric attributes (REAL, DREAL, COMPLEX, DCOMPLEX, INTEGER, INDEX, INDEX_1, DECIMAL) the literal must be numeric or a numeric named constant. When using INITIAL_VALUE at the group level, rules for moving group values must be followed, refer to Section 12.1.3.

For CHARACTER attributes, a non-numeric literal enclosed in single quotes, or the non-numeric named constants, e.g., SPACES, may be used after INITIAL_VALUE. In the case of STATUS attributes, the literal must be one of the status words for that attribute. In the case of RULE attributes, the literal must be one of the rule_names for that attribute. When used with an ALIAS clause for an elementary attribute, the INITIAL_VALUE clause must come after the ALIAS clause, on a separate line. See examples below

INITIAL_VALUE may also be used as a qualifying clause within a structure that is subsequently *redefined*. However it may not be used after a REDEFINES clause is used. Refer to the REDEFINES example in Section 11.5.3.

COMPLEX and DCOMPLEX numbers can be initialized using a complex pair (real_number, imaginary_number), or the REDEFINES clause to treat them as pairs of REAL and DREAL numbers.

INITIAL VALUE examples

```
AIRCRAFT_DATA
1  AIRCRAFT_NUMBER          INTEGER INITIAL_VALUE 1
1  S_N_RATIO                REAL    INITIAL_VALUE 3.2
1  USER_INPUT_2            DECIMAL Z(3)9.9  INITIAL_VALUE 6.1

1  TOTAL_POWER              COMPLEX INITIAL_VALUE (30.06, 42.5)

1  LOCAL_POWER
2  REAL_POWER               REAL INITIAL_VALUE 28.44
2  IMAGINARY_POWER          REAL INITIAL_VALUE 21.56
1  COMPLEX_POWER REDEFINES LOCAL _POWER COMPLEX

1  AIRCRAFT_MESSAGE          CHAR 50 INITIAL_VALUE SPACES

1  AIRCRAFT_POSITION          STATUS GROUNDED, IN_FLIGHT
                              INITIAL_VALUE GROUNDED

1  AIRCRAFT_POSITION          CHAR 9
                              ALIAS GROUNDED          VALUE 'GROUNDED '
                              ALIAS IN_FLIGHT          VALUE 'IN_FLIGHT'
                              INITIAL_VALUE 'GROUNDED '
```

INITIAL_VALUE may *not* be used with QUANTITY clauses or subordinate to QUANTITY. One can, however initialize a whole group of DECIMAL or CHARACTER data array if a top-level attribute is created, for example:

```
TOP_LEVEL_GROUP          INITIAL_VALUE ZERO
1  NEXT_LEVEL QUANTITY(3)
2  DEC_ATTRIBUTE          DECIMAL 9(2)
```

INITIAL_VALUE Clause – Creating Complex Tables

By defining INITIAL_VALUES as shown in Figure 11-2, one does not have to search through the rules to see where initial values are set. This helps to ensure proper initialization, since the designer simply checks the attributes in the Resource.

RESOURCE NAME: TRANSCEIVER			
TRANSCEIVER_INSTANCES			
1	TRANSMITTER	INDEX	
1	RECEIVER	INDEX	
GENERAL_PARAMETERS			
1	TRANSMITTER_POWER	REAL	INITIAL_VALUE 100
1	RECEIVER_THRESHOLD	REAL	INITIAL_VALUE 120
RADIO QUANTITY(500)			
1	TRANSCEIVER	STATUS	TRANSMITTING RECEIVING IDLE OFF
1	LOCATION		
2	X_POSITION	REAL	
2	Y_POSITION	REAL	
2	ELEVATION	REAL	
1	ANTENNA_HEIGHT	REAL	
1	ANTENNA_GAIN	REAL	
RECEIVER_CONNECTIVITY_VECTOR QUANTITY(500)			
1	POWER_AT_RECEIVER	REAL	
1	TOTAL_NOISE_POWER	REAL	
1	CONNECTIVITY_MATRIX	QUANTITY(500)	
2	PROPAGATION_LOSSES		
3	TERRAIN_LOSS	REAL	
3	FOLIAGE_LOSS	REAL	
3	TOTAL_LOSS	REAL	
2	SIGNAL_POWER	REAL	
2	SIGNAL_TO_NOISE_RATIO	REAL	
2	LINK_DELAY	REAL	
2	LINK	STATUS	GOOD FAIR POOR
TRANSCEIVER_RULES			
1	TRANSCEIVER_PROCESS	RULES	GOOD_RECEPTION CONFLICTING_RECEPTION CONFLICTING_BROADCAST

Figure 11-2. Example of hierarchical tables.

CHAPTER 12.

PROCESS DESCRIPTIONS

CREATING AND MANAGING TRANSFORMATIONS

In the CAD system described here, a *process* contains the instructions that act on data as transformations. At first glance, a process may look like a *function* or *subroutine* in a typical programming language. However, many of the problems we deal with today are dominated by sets of rules that operate on complex data structures, not just mathematical functions. This implies many large dissimilar transformations that must be tailored to the complex data structures upon which they operate. Although the VisiSoft process language has properties similar to CMS-2 (developed by Grace Hopper), it is an obvious departure from all programming languages, especially those in use today.

AN ELEGANT SOLUTION TO FLOW OF CONTROL

Nesting of control structures is a feature of virtually all conventional programming languages. For example, it is not uncommon to see

```
An if-statement
    containing an if-statement
        which contains a while-loop
```

Such an example by itself is not especially problematic, but does suggest the mental complexity of keeping track of code with nested control. Moreover, the mental complexity increases as the length of the code and the length of nested sequences grows. When using a poor language, it is not uncommon for single blocks of code to extend over more than one page

Things can get complex even without a nested loop. When there is nesting and the statements contained in the IF are of some length, getting a clear picture of the entire structure is difficult. When nested IFs cover many lines, the logic becomes hard to understand.

We propose a fresh look at something so commonplace that we take it for granted, that is, flow of control. To control the complexity of highly conditional transformations, we must again make use of hierarchies and be able to deal easily with multiple layers of hierarchy. One approach is using nested IFs. The VisiSoft approach is to group statements into RULEs within processes. This solves multiple problems. An example is shown in Figure 12-1.

Each process is decomposed into rules that share the same resources as the process. The hierarchical rule facility is provided through a simple *one-in, one-out* control structure embodied in the EXECUTE statement, which takes on various forms. This statement allows the designer to deal with rules that are at an "equal level" in the hierarchy of logical operations, without resorting to the dangers of GOTO statements. The control structures are strictly *one-in, one-out*. We note that "one-in one-out" implies control is transferred to a rule, then transferred back to the statement following the one that invoked the rule.

Controlling Complexity With Rule Hierarchies

A collection of statements that is given a name is called a "*rule*". Figure 12-1 shows an example of a process that has six rules. Generally a process will consist of a number of rules. Rules that are specific to an algorithm's data structures are all usually contained in the same spot, are easier to build, and much easier to understand during the support years.

Each process has a top-level rule, e.g., PLACE_CALL in Figure 12-1. When the statements in the top-level rule have been executed, control returns to the calling process. Any rule may contain EXECUTE statements that invoke other rules within the process.

```
PLACE_CALL                                     level 1
  IF CLOCK_TIME IS GREATER THAN ONE_HOUR
    STOP.
  IF ACTIVITY IS WAITING_TO_CALL
    EXECUTE ATTEMPT_CALL
  ELSE EXECUTE RETRY_LATER.

-----
ATTEMPT_CALL                                   level 2
  INCREMENT CALLS_ATTEMPTED
  IF LINES_IN_USE(OFFICE) ARE LESS THAN
    LINES_IN_OFFICE(OFFICE)
    EXECUTE MAKE_CALL
  ELSE EXECUTE BLOCK_CALL.

RETRY_LATER
  SET ACTIVITY(SOURCE) TO RETRY_LATER
  CALL TERMINATE_CALL

-----
MAKE_CALL                                     level 3
  INCREMENT LINES_IN_USE(OFFICE)
  IF CALLERS_PLAN(SOURCE) IS PLACE_NEW_CALL
    SET PHONE_NUMBER TO UNKNOWN
    EXECUTE LOOK_UP_NUMBER UNTIL PHONE_NUMBER IS FOUND.
  OFFICE_NUMBER = OFFICE(DESTINATION)
  CALL CONNECT_CALL

BLOCK_CALL
  INCREMENT CALLS_BLOCKED
  SET SIGNAL_TO_SUBSCRIBER TO BUSY
  MOVE 'BLOCKED AT SOURCE' TO CALL_STATE_OUTPUT

-----
LOOK_UP_NUMBER                               level 4
  DESTINATION = (TOTAL SUBSCRIBERS * RANDOM) + 1
  IF DESTINATION IS NOT EQUAL TO SOURCE
    SET PHONE_NUMBER TO FOUND.
```

Figure 12-1. Example of a Process.

The example of Figure 12-1 involves four levels of hierarchical control, yet is simple to understand. It also shows the ability to "push down" the complexity of rule sets into a hierarchy of logical levels. As a result, a process is typically somewhat larger (containing much more logic) than "well written" C++ or Java functions that are more typically the size of a rule. But it should be much more understandable, and will require many fewer comments, most often none. A process with 20 rules may take 8 to 10 C-based language functions to implement.

This dual structure has several advantages:

1. At the end of a rule, control returns to the statement following the EXECUTE statement that invoked that rule. This guarantees the 1-in, 1-out property.
2. Flow of control is linear within a rule.
3. A process may contain one or more rules, each identified by a name. This provides flexibility in the number of conditional statements that a process can support.
4. One may EXIT a rule at any time, e.g.,

```
IF SOME_CONDITION IS SATISFIED
    EXIT THIS RULE .
```
5. There is no need for nesting of IF statements.
6. Except for the first rule, rules may be placed in any order.
7. There is no recursion of rules.

Understandability of Complex Conditional Situations

One of the most important benefits of this approach is the simplicity of complex conditional situations. Consider the following example:

```
IF SYMBOL IS AN UNDERSCORE
OR SYMBOL IS A PERIOD
    EXECUTE CHECK_WORD_BLOCK
ELSE
    EXECUTE SCAN_FOR_SPECIAL_CASES.

IF STATEMENT IS A SPECIAL_CASE
    EXIT THIS RULE
ELSE ...
```

Here we see the equivalent of the case statement. However, the statements that are normally contained within a case statement may now be placed in a separate rule, in this case CHECK_WORD_BLOCK and SCAN_FOR_SPECIAL_CASES. This adds great clarity, as we can read and understand the higher level of control without being distracted by nested details in the next level that themselves may be quite complex.

Figure 12-1 provides an example of a process structure that follows the rule for grouping hierarchical logical levels. The built-in rules eliminate strings of call statements that cause unstable designs because of the complex dependencies created. They also run faster. And, since a large set of rules shares only the data available to the process, they also support ease of use of parallel processors. Since the *logical levels are totally independent of position*, the process may be organized in any manner the designer deems most understandable. Except for the first rule appearing first, the rest of the rules may be shuffled like a deck of cards.

The largest benefits of the hierarchical rule structure of processes are the understandability of complex conditional statements, and the ease with which one can add new conditions as the software is enhanced. Additional features of the process language, particularly the EXIT THIS RULE statement, serve to flatten conditions within a rule, making the logic apparent. This eliminates nested IF statements as well as call strings to small fragments of code that are used as the typical alternative.

Understandability of Loop Structures

A related property of the approach to control structures is the isolation of the body of a loop (the statements to be repeated) in a separate named rule. Only the rule name is used within the control structure itself.

Thus we may write something like

```
EXECUTE LOOK_UP_NUMBER
  UNTIL PHONE_NUMBER IS FOUND
```

and place the body of the loop elsewhere

```
LOOK_UP_NUMBER
  DESTINATION = (TOTAL_SUBSCRIBERS * RANDOM) + 1
  IF DESTINATION IS NOT EQUAL TO SOURCE
    SET PHONE_NUMBER TO FOUND.
  . . .
```

After the LOOK_UP_NUMBER rule is executed, control automatically returns to the EXECUTE statement. Here again, rather than a sequence of nested structures, we can easily understand the control since it is at a single level.

Rule Pointers

Another step towards speed and understandability of processes is the ability to assign the name of a rule to a “rule pointer”.

A RULE clause is used to define the allowed rule names that a rule pointer can assume during execution. Consider:

```
NEXT_ACTION      RULE  INITIALIZE_NETWORK,
                      START_TRANSMISSION,
                      START_RECEPTION,
                      DISCONNECT_CALL
```

Here, NEXT_ACTION defines a set of allowed rules.

The rule pointer NEXT_ACTION will likely be set in a conditional statement prior to a point where the rule is to be executed, such as:

```
IF TRANSCEIVER(TRANSMITTER) IS TRANSMITTING
  SET NEXT_ACTION TO START_TRANSMISSION
ELSE IF TRANSCEIVER(RECEIVER) IS RECEIVING
  SET NEXT_ACTION TO START_RECEPTION
```

Later one can then simply have.

```
EXECUTE NEXT_ACTION
```

This mechanism provides a direct, and therefore fast, transfer of control similar to the computed GOTO in FORTRAN. The improvement of this approach is twofold. First is the direct return of control to the next statement (one-in one-out versus GOTO). Second, the name of the rule, e.g., START_RECEPTION is used in the set statement, instead of a number - making it clear what the pointer is being used for. As above, meaningful names are used and the choice of action can be set when an appropriate condition is met.

Process Pointers

The PROCESS pointer clause is similar to the RULE pointer clause. It is used to define each of the allowed process names that a PROCESS pointer can assume during execution. It is used to support the PROCESS pointer version of the CALL statement for executing processes.

The mechanism is almost identical to rule pointers. For example, the process pointer is defined in a resource:

```
NEXT_PROCESS      PROCESS COMPUTE_TIMERS,  
                  DRAW_TERRAIN,  
                  COMPUTE_MEASURES
```

It can then be used in a process as shown below.

```
IF INPUT_OPTION IS INITIATE  
    SET NEXT_PROCESS TO COMPUTE_TIMERS  
ELSE IF INPUT_OPTION IS CALCULATE  
    SET NEXT_PROCESS TO DRAW_TERRAIN  
.  
.  
.  
CALL NEXT_PROCESS
```

Chapter 7 set out the information required to describe the architecture for a process, the second language of VisiSoft. The internal process description consists of statements, which are grouped into hierarchies of rules.

When a software language provides the ability to define large data hierarchies, a number of important capabilities are facilitated. However, recognition of the value of these facilities takes time to measure and appreciate. After many years of experience working with large teams of engineers building complex systems, the contribution of these facilities to the critical time factors becomes obvious. Some of these are offered below.

- Fast run times due to the ability to move complex data structures using a single instruction fetch.
- The ability to access elements within a complex data structure - where it is clear where each fits within the organization (like standing on a parade ground).
- Small processes are useful. But without the ability to build large representative hierarchies of both data and instructions, one is forced into small routines for everything, eliminating the ability to build obviously simplified instruction hierarchies. Recognition is the key to understandability, conquering size. Having to read and change many routines to modify a complex algorithm can be greatly simplified with a large but obvious hierarchy.
- Simplifying architectures by reducing the number of processes and resources in a module, and also the number of modules in a system.

Statements and rules form a hierarchical structure for describing processes, as follows:

- Each process may consist of one or more rules, each with a unique name. The rule name is any GSS word (other than the reserved words listed in the GSS or VSE User Manuals, see [67] or [150]).
- Each rule name must appear on a separate line followed by the statements, which make up the rule.
- Each statement must begin on a new line, but can extend over many lines.
- All process statements must begin in column 5 or beyond, except for rule names, which must begin in columns 1 through 4. The maximum line length is 72 characters.

When a process is invoked, the first rule is executed first, starting with the first statement. Other rules within this process may be executed by using an EXECUTE statement which alters the flow of control (see Section 12.3.1). Execution of a process terminates once the last statement in the first rule is performed. This control structure means that the ordering of rules within a process is unimportant, and they may therefore be arranged for ease of readability. The only exception is that the main rule, which begins the process, must be written first.

Figure 12-1 shows an example of the process PLACE_CALL, which has six rules. The statements, which describe a process, may be drawn from any of four statement types:

- (1) Assignment statements
- (2) Conditional statements
- (3) Control statements
- (4) Input/Output (I/O) statements

Each of these is described in the sections below.

12.1 ASSIGNMENT STATEMENTS

Assignment statements instruct GSS to move data, assign numeric values, or assign states to specified resource attributes. Different forms of the assignment statement are used for the different types of attributes. For numeric attributes (those with INTEGER, REAL, DREAL, COMPLEX, DCOMPLEX, INDEX or INDEX_1 qualifying clauses) use a numeric assignment statement or a MOVE statement; for data (CHARACTER or DECIMAL) attributes, use a MOVE statement; for STATUS attributes, use a SET statement. Details on these various forms of the assignment statement follow.

12.1.1 NUMERIC ASSIGNMENT STATEMENTS

Scientists and engineers will appreciate the fact that VisiSoft arithmetic statements are fashioned after FORTRAN, a well-designed language for writing equations. The exception is the use of multiple subscripts, where a common complaint in FORTRAN is the ordering of the subscripts. In VisiSoft, they follow the typical matrix notation with the right most iterator being first. When assigning values to numeric resource attributes, the assignment symbol (=) is used.

$$\text{attribute_name_1} \quad [,\dots\text{attribute_name_n}] \quad = \quad \left\{ \begin{array}{l} \text{numeric_literal} \\ \text{ZERO (E) (S)} \\ \text{arithmetic_construct} \\ \text{attribute_name} \end{array} \right\}$$

More than one attribute may appear to the left of the assignment symbol. Arithmetic constructs are formed by combining attribute names, numeric literals, arithmetic operators, built-in functions, and parentheses to represent computations to be performed before assigning a value to the attribute(s) named on the left of the assignment symbol. The *arithmetic operators* are:

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

VisiSoft also recognizes the unary operators + and - , denoting multiplication by +1 and -1, respectively. When used in this sense, the operators and their single operand must be enclosed in parentheses. For example, the product of attribute A and the negative of attribute B must be written as A * (-B).

Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first. When expressions are contained within a nest of parentheses, evaluation proceeds from the least inclusive (innermost) to the most inclusive (outermost) set.

When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order is implied:

1. **
2. * and /
3. + and -

Parentheses are used either to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear, or to modify the normal hierarchical sequence of execution, in expressions where it is necessary to have some deviation from the normal precedence. When the order of consecutive operations on the same hierarchical level is not completely specified by parentheses, the order of operation is from left to right.

When numeric assignment operations are performed, decimal points (whether explicitly or implicitly defined) are automatically aligned first.

Numeric assignments may be of the form:

- (1) Deterministic - represented by a positive real numeric literal or resource attribute value, or
- (2) Probabilistic - represented by a statistical distribution with specified parameters.

Numeric assignment examples

```
CALL_DURATION = TNORMAL(AVERAGE_CALL_DURATION, 1)
S_N_DISCRIMINATION(TRANSCIEVER_NUM) = .015
X_DIR,Y_CHANGE(1),X_CHANGE(3) = 1
AIRCRAFT_STATE = ZERO
PERCENT_BUSY = (TOTAL_BUSY_CALLS * 100)/TOTAL_CALLS
TOTAL_CALLS = TOTAL_CALLS + 1
POS_RESULT = (-NEG-RESULT)
LOG_TERM = 10**(NOISE_POWER*7.1/.03)
ANSWER = TABLE_VALUE(M+1, N(M+1))
```

12.1.1.1 Built-In Functions

Built-in functions are special (reserved) words, which represent intrinsic numeric functions contained in what is typically referred to as a set of scientific subroutines. A complete list of the built-in functions with explanations of their meanings is given in the GSS or VSE User's Manuals. Some Built-in function examples are ATAN() and MAX():

```
AZIMUTH_ANGLE = ATAN(Y_COORDINATE/X_COORDINATE)
BUSY_CALLS = MAX(CALLS(LOCAL), CALLS(FOREIGN))
```

12.1.1.2 Complex Functions

Complex functions are supported in GSS in assignment statements and arithmetic constructs. Addition, subtraction, multiplication, division, and power are translated automatically for complex numbers. Additional built-in functions can be used in complex algebraic expressions. Built-in functions such as ABS, EXP, LN, and SQRT will depend upon whether the argument is real or complex. The rules for complex arguments are given below.

Complex Number: COMPLEX(a, b)

A complex number, z , can be defined by its real (a) and imaginary (b) parts.

$z = \text{COMPLEX}(a, b) = a + ib$, where i is the imaginary operator $\sqrt{-1}$.

Real Value: REAL(z)

The real value of a complex number is the value of the real part of the complex number.

$\text{REAL}(z) = a$, where $z = a + ib$

Imaginary Value: IMAG(z)

The imaginary value of a complex number is the value of the imaginary part of the complex number.

$\text{IMAG}(z) = b$, where $z = a + ib$

Conjugate: CONJ(z)

The conjugate of a complex number is the complex number with the sign of the imaginary part reversed.

$\text{CONJ}(z) = a - ib$, where $z = a + ib$

Absolute Value: ABS(z)

The absolute value of a complex number is the positive square root of the sum of the squares of the real and imaginary parts.

$$|z| = \sqrt{a^2 + b^2}$$

Exponential: EXP(z)

The exponential (e) raised to a complex number is given as follows.

$$e^z = e^{(a + ib)} = e^a * e^{ib}, \quad \text{where } e^{ib} = \cos(b) + i \sin(b)$$

$$e^z = e^a * (\cos(b) + i \sin(b)) = e^a * \cos(b) + i e^a * \sin(b)$$

Natural Log: LN(z)

The natural log of a complex number is, in general, a multi-valued function given as follows.

$$\ln(z) = \ln(a + ib) = \ln(|z|) + j(\theta + 2n\pi),$$
$$\text{where } \theta = \arctan\left(\frac{b}{a}\right), \text{ and } n = 0, \pm 1, \pm 2, \dots$$

We will use the principal value, defined as:

$$\mathbf{LN}(z) = \ln(|z|) + i\Theta, \quad \text{where } -\pi < \Theta \leq \pi$$

Power: z**u

A complex number, z, raised to a power, u (a real number), is given as follows:

$$z^u = e^{\ln(z^u)}$$
$$= e^{u \ln(z)}, \quad \text{where } \ln(z) = \mathbf{LN}(z) = \ln(|z|) + i\Theta,$$
$$\text{and } \Theta = \arctan\left(\frac{b}{a}\right)$$

then,

$$z^u = e^{u \ln(|z|)} * e^{iu\Theta}$$
$$= |z|^u * [\cos(u\Theta) + i \sin(u\Theta)]$$

thus,

$$z^u = |z|^u * \cos(u\Theta) + i |z|^u \sin(u\Theta) .$$

12.1.2 INCREMENT, DECREMENT, ADD, SUBTRACT STATEMENTS

Four special keywords exist for the common operation of adding or subtracting a positive constant value. The words INCREMENT, DECREMENT, ADD, and SUBTRACT are used as follows:

$$\left\{ \begin{array}{l} \text{INCREMENT} \\ \text{DECREMENT} \end{array} \right\} \text{ attribute_name_1 } [, \dots \text{ attribute_name_n}]$$
$$\left[\text{BY } \left\{ \begin{array}{l} \text{attribute_name} \\ \text{unsigned_numeric_literal} \end{array} \right\} \right]$$
$$\text{ADD } \left\{ \begin{array}{l} \text{attribute_name} \\ \text{unsigned_numeric_literal} \end{array} \right\} \text{ TO attribute_name_1 } [, \dots \text{ attribute_name_n}]$$
$$\text{SUBTRACT } \left\{ \begin{array}{l} \text{attribute_name} \\ \text{unsigned_numeric_literal} \end{array} \right\} \text{ FROM attribute_name_1 } [, \dots \text{ attrib_name_n}]$$

When the INCREMENT or DECREMENT keywords are used, the value to be added to or subtracted from the named attribute may be omitted. If so, the default is an increment (decrement) of 1.

Examples

```
DECREMENT TOTAL_LOSS BY 100
INCREMENT CHANGE_NUM
DECREMENT TOTAL_LOSS BY PART_SUM
INCREMENT DAY_COUNT BY 7
INCREMENT X_COUNT, Y_COUNT
ADD 100.3 TO TOTAL_LOSS
SUBTRACT PART_SUM FROM TOTAL_LOSS
```

12.1.3 MOVE STATEMENT

The MOVE statement is used to move data or assign values to data or numeric attributes. It is particularly useful when assigning values to a group attribute, which may contain a mixture of attribute types.

$$\text{MOVE } \left\{ \begin{array}{l} \text{resource_name_1} \\ \text{attribute_name_1} \\ \text{nonnumeric_literal} \\ \text{named_constant} \\ \text{numeric_literal} \end{array} \right\} \text{ TO } \left\{ \begin{array}{l} \text{resource_name_2} \\ \text{attribute_name_2 } [, \dots \text{ attribute_name_n}] \end{array} \right\}$$

The movement of nonnumeric data to numeric attributes is not allowed. The move statement is subject to the following automatic editing rules.

When the receiving area of a MOVE statement is a *numeric* attribute (REAL, DREAL, INTEGER, INDEX, or INDEX_1) or a *decimal* attribute:

- Decimal points will be aligned and digits of the sending number will be truncated at either end, as required by the size of the receiving area. Any receiving number digit not covered by the sending number will be filled with zeros. When there is no explicit decimal point (as in INTEGER attributes) the decimal point is 'implied' to be in the right-most position. No sign implies a positive number.
- When a numeric attribute is moved to a DECIMAL attribute, zeros are changed to spaces when zero suppression (Z) is specified. When moving a decimal field to a numeric attribute, zeros are assumed present in the Z fields.

When the receiving area of a MOVE statement is a *data* attribute (CHARACTER, STATUS, or RULE):

- Data is aligned on the left and is either truncated at the right or filled with spaces to match the size of the receiving area.
- When moving a STATUS attribute to another STATUS attribute, the sending status names must be a subset of the receiving status names.

Movement of a group level attribute is treated the same as a character data move, with no regard for the elementary data types within the sending (or receiving) field.

Table 12-1 Allowed sending and receiving attribute types in a MOVE statement.

SENDING FIELD	ALLOWED RECEIVING FIELD		
	CHARACTER	NUMERIC	DECIMAL
CHARACTER DATA	YES	NO	NO
NUMERIC DATA	NO	YES	YES
DECIMAL DATA	YES	YES	YES
NAMED CONSTANT			
SPACE	YES	NO	NO
LOW_VALUE	YES	NO	NO
HIGH_VALUE	YES	NO	NO
ESCAPE_CHARACTER	YES	NO	NO
LINE_FEED_CHARACTER	YES	NO	NO
CARRIAGE_RETURN	YES	NO	NO
ZERO	YES	YES	YES
PI	NO	YES	YES
NONNUMERIC LITERAL	YES	NO	NO
NUMERIC LITERAL	NO	YES	YES

VSE/FIGUR 9-1 As of 11/15/05

MOVE examples

Example 1.

```
MOVE ATTENUATION_FACTOR TO STORED_NUMBER
```

where:

```
ATTENUATION_FACTOR is REAL and STORED_NUMBER is DEC 9(2).9(3)
```

Attribute	Before move	After move
ATTENUATION_FACTOR	3.276000E-1	3.276000E-1
STORED_NUMBER		00.327

Example 2.

```
MOVE TOTAL_CALLS TO SUMMARY_VALUE
```

where:

```
TOTAL_CALLS is INTEGER and SUMMARY_VALUE is DEC 9(5)
```

Attribute	Before move	After move
TOTAL_CALLS	55231	55231
SUMMARY_VALUE		55231

12.1.4 CONVERT STATEMENT

The CONVERT statement is used to convert CHARACTER attributes containing valid numbers to numeric attributes.

```
CONVERT numeric_literal_name TO internal_number_name
      ON_ERROR statement
```

When using CONVERT, numeric_literal_name is a CHARACTER field of not more than 24 characters, and internal_number_name may be an INDEX_1, INDEX, INTEGER, REAL, or DREAL number. Errors are handled using the ON_ERROR statement. In the case of an error, the internal_number_name is not updated. The Content in the example below indicates the possible numeric representations.

Examples

NUMBER_X_COLS	CHAR 6	Content: ' 999 '
INPUT_FLOAT	CHAR 8	Content: '+999.99 '
INPUT_EXPO	CHAR 12	Content: ' +.999E+99'
INPUT_NUMBER	CHAR 16	Content: ' +.999E+99 '
COLUMNS		INDEX
REAL_FLOAT		REAL
REAL_EXPO		REAL
INTERNAL_NUMBER		DREAL

More Examples

```
CONVERT NUMBER_X_COLS TO COLUMNS
ON_ERROR EXECUTE NUMERIC_ERROR_RULE

CONVERT INPUT_FLOAT TO REAL_FLOAT
ON_ERROR EXECUTE NUMERIC_ERROR_RULE

CONVERT INPUT_EXPO TO REAL_EXPO
ON_ERROR EXECUTE NUMERIC_ERROR_RULE

CONVERT INPUT_NUMBER TO INTERNAL_NUMBER
ON_ERROR EXECUTE NUMERIC_ERROR_RULE
```

12.1.5 SET STATUS STATEMENT

The SET STATUS statement is used to set the state of resource attributes defined by the STATUS clause. It is another major contributor to understandability of complex algorithms, making it much easier to understand why an attribute is being set, as well as simplify complex conditional statements.

```
SET attribute_name [STATUS] TO status_name
```

Status-name must be one of the status conditions associated with the named attribute(s). It must be a VSE word. This provides an understanding of how a particular conditional variable is set.

SET STATUS examples

```
SET TRANSCEIVER(RECEIVER) TO IDLE
```

where TRANSCEIVER is defined as follows:

```
TRANSCEIVER_DATA  QUANTITY(50)
1 TRANSCEIVER      STATUS IDLE
                    TRANSMITTING
                    RECEIVING
                    BUSY
```

12.1.6 SET ALIAS STATEMENT

The SET ALIAS statement is used to set the state of resource attributes as defined by the ALIAS clause.

```
SET attribute_name [ALIAS] TO alias_name
```

Alias-name must be a unique alias condition associated with the named attribute, i.e., it cannot have multiple values. It must be a VSE word. This provides an understanding of how a particular aliased variable is set.

SET ALIAS example

```
SET MONTH TO JUNE
```

where MONTH is defined here as a single byte integer as follows:

CALENDAR_INFORMATION		INDEX_1
1	MONTH	
	ALIAS MARCH	VALUE 3,
	ALIAS JUNE	VALUE 6,
	ALIAS SEPTEMBER	VALUE 9,
	ALIAS DECEMBER	VALUE 12

12.1.7 SET COLOR STATEMENT

The SET COLOR statement is used to set the color of graphical objects that have been defined using the COLOR qualifying clause. Refer to Chapter 11, Section 11.4.3.3, and Chapter 2 of the Run-Time Graphics (RTG) User's Manual.

```
SET object_color [COLOR ] TO color_name[(ramp)]
```

Color_name must be one of the color names defined with the color attribute. Having performed the SET statement shown above, the object will be displayed with the selected color when drawn on the screen. In addition, the object_color can be tested in a conditional statement (refer to Section 12.2.1.3). A color attribute is 24 bytes and you can set the color attribute to a color with color ramp. The default ramp number is 32.

SET COLOR examples

```
SET LINE_COLOR TO RED
SET CONNECTING_LINK COLOR TO GREEN(22)
```

12.1.8 SET RULE STATEMENT

The SET RULE statement is used to set the state of rule-pointer attributes that have been defined using the RULE qualifying clause.

```
SET rule_pointer [RULE] TO rule_name
```

Rule_name must be one of the rule names associated with the rule_pointer attribute. Having performed the SET statement shown above, "rule_name" will be executed when an EXECUTE RULE_POINTER statement is encountered (refer to Section 12.3.1). In addition, the rule_pointer can be tested in a conditional statement (refer to Section 12.2.1.4).

SET RULE examples

```
SET CURRENT_SECTION RULE TO CONTROL_SECTION
SET CONTROL_POINTER RULE TO PHASE_1
```

CONTROL_SECTION must be one of the rules that can be executed in a statement of the form EXECUTE CURRENT_SECTION RULE. PHASE_1 must be one of the rules that can be executed in a case statement of the form EXECUTE CONTROL_POINTER RULE.

12.1.9 SET PROCESS STATEMENT

The SET PROCESS statement is used to set the state of process_pointer attributes that have been defined using the PROCESS qualifying clause (refer to Section 11.4.3.5).

```
SET process_pointer [PROCESS] TO process_name
```

Process_name must be one of the process names associated with the process_pointer attribute. Having performed the SET statement shown above, "process_name" will be executed when an CALL PROCESS_POINTER statement is encountered (refer to Section 12.2.6). In addition, the process_pointer can be tested in a conditional statement (refer to Section 12.2.1.5).

SET PROCESS examples

```
SET NEXT_PROCESS TO COMPUTE_PERFORMANCE  
SET PROCESS_POINTER PROCESS TO PHASE_1
```

COMPUTE_PERFORMANCE must be one of the processes listed in the NEXT_PROCESS attribute statement defined in a resource. PHASE_1 must be one of the processes listed in the PROCESS_POINTER attribute statement defined in a resource.

12.1.10 SET EVENT Statement

The SET EVENT statement is used to set the event_states that have been defined using the EVENT qualifying clause.

```
SET event_name [EVENT] TO event_state
```

Event_name must be an EVENT attribute with corresponding defined event_states. Having performed the SET statement shown in the Format box, the value of the event_name EVENT will be set to that of the specified event_state. Processes sharing the event_name EVENT will be checked by the OS to determine if they are in a WAIT state. If so, and if the wait state is to be terminated upon the newly changed event_state, then the value of the new event_state will be put into that process's OS resource, and the process will continue with the next statement following the WAIT UNTIL statement (see Section 12.7.1 below).

SET EVENT examples

```
SET MY_EVENT_STATE TO GO  
SET MY_EVENT_STATE TO STOP
```

12.2 CONDITIONAL STATEMENTS

Conditional statements instruct VSE to take different actions depending on the circumstances that exist when the conditional statement is encountered. Conditional statements are used to form the most complex algorithms in any language. It is imperative that they are easily understood, particularly by subject area experts that may be the only people that understand the effect of conditions to be achieved. It is interesting to watch people explain conditional statements in difficult to understand programming languages. The explanations are almost always done in English.

Direct readability depends upon the ability to describe complex conditions easily. To do this requires different constructs that support the expression of different types of conditions. Most importantly, the rule hierarchies eliminate the need for nested IFs, i.e., an IF statement inside of an IF statement. We note that CASE statements are not nested IFs.

The general format of the conditional statement is:

$$\text{IF [NOT] condition [THEN] } \left\{ \begin{array}{l} \text{statement_1 } [\dots \text{statement_n}] \text{ EXIT [THIS] RULE} \\ \text{NEXT STATEMENT} \end{array} \right\}$$
$$\left[\text{ELSE } \left\{ \begin{array}{l} \text{statement_1 } [\dots \text{statement_n}] \text{ EXIT [THIS] RULE} \\ \text{NEXT STATEMENT} \end{array} \right\} \right]$$

Each statement begins on a new line and different parts of the statement are separated by one or more spaces. Each part of the conditional statement format is now described.

12.2.1 CONDITIONAL CONSTRUCTS

A conditional construct is an expression that, taken as a whole, may be either true or false, depending on the circumstances existing when the expression is evaluated. They are designed to make the language understandable. The following conditional constructs are available in VisiSoft:

- (1) Class condition
- (2) Status condition
- (3) Color condition
- (4) Rule condition
- (5) Process condition
- (6) Alias condition
- (7) Event condition
- (8) Relation condition
- (9) Sign condition
- (10) File condition
- (11) Compound condition
- (12) Simulation condition
- (13) Optimization condition

12.2.1.1 CLASS Condition

The class test determines whether the current value of a CHARACTER or DECIMAL resource is composed of alphabetic characters (A-Z or space) or is numeric (digits 0 through 9).

$$\text{attribute_name} \begin{bmatrix} \text{IS} \\ \text{ARE} \end{bmatrix} [\text{NOT}] \left\{ \begin{array}{l} \text{NUMERIC} \\ \text{ALPHABETIC} \end{array} \right\}$$

The clarity words, IS or ARE, are optional in this format. The keyword NOT may optionally be included to test the converse condition.

12.2.1.2 STATUS Condition

In a status condition, a status attribute is tested to determine if its state is that of the specified status name. The status_name must be one of the allowed states associated with the named status attribute. The clarity words STATUS, IS, ARE, A, or AN may also be included. The keyword NOT may optionally be included to test the converse condition.

$$\text{status_attribute} [\text{STATUS}] \begin{bmatrix} \text{IS} \\ \text{ARE} \end{bmatrix} [\text{NOT}] \begin{bmatrix} \text{A} \\ \text{AN} \end{bmatrix} \text{status_name}$$

Status condition examples

```
TRANSCEIVER IS TRANSMITTING  
TELEPHONE IS NOT BUSY
```

12.2.1.3 COLOR Condition

In a color condition, a color attribute is tested to determine if its color is that of the specified color_name.

$$\text{color_attribute} [\text{COLOR}] \begin{bmatrix} \text{IS} \\ \text{ARE} \end{bmatrix} [\text{NOT}] \begin{bmatrix} \text{A} \\ \text{AN} \end{bmatrix} \text{color_name}$$

The color_name must be one of the allowed colors associated with the named color attribute. The clarity words COLOR, IS, ARE, A, or AN may also be included. The keyword NOT may optionally be included to test the converse condition.

Color condition examples

```
RADIO_LINK COLOR IS GREEN  
FLAG IS NOT RED
```


12.2.1.4 RULE Condition

In a rule condition, a rule_pointer attribute is tested to determine if its state is that of the specified rule name.

```
Rule_pointer [RULE] IS [NOT] rule_name
```

Rule_name must be one of the allowed rules associated with the rule_pointer attribute defined using the RULE qualifying clause. The clarity words RULE may also be included. The keyword NOT may optionally be included to test the converse condition.

Rule condition examples:

```
CURRENT_SECTION IS CONTROL_SECTION  
NEXT_RULE IS NOT FINAL_STEP
```

CONTROL_SECTION must be a rule listed in the CURRENT_SECTION attribute statement defined in a resource. FINAL_STEP must be a rule listed in the NEXT_RULE attribute statement defined in a resource.

12.2.1.5 PROCESS Condition

In a process condition, a process_pointer attribute is tested to determine if its state is that of the specified process name.

```
Process_pointer [PROCESS] IS [NOT] process_name
```

Process_name must be one of the allowed processes associated with the process_pointer attribute defined using the PROCESS qualifying clause (refer to Section 11.4.3.5.). The clarity word PROCESS may also be included. The keyword NOT may optionally be included to test the converse condition.

PROCESS condition examples:

```
CURRENT_PROCESS IS CONTROL_PROCESS  
NEXT_PROCESS IS NOT FINAL_PROCESS
```

CONTROL_PROCESS must be a process listed in the CURRENT_PROCESS attribute statement defined in a resource. FINAL_PROCESS must be a process listed in the NEXT_PROCESS attribute statement defined in a resource.

12.2.1.6 ALIAS Condition

In an alias condition, a resource attribute is tested to determine if its value is one of a group collectively identified by a specified alias name.

```
attribute_name  $\begin{bmatrix} \text{IS} \\ \text{ARE} \end{bmatrix}$  [NOT]  $\begin{bmatrix} \text{A} \\ \text{AN} \end{bmatrix}$  alias_name
```

The alias_name must be associated with the named attribute via an ALIAS qualifying clause in the resource (see Section 11.5.2). The clarity words IS, ARE, A, and AN are optional in this format. The keyword NOT may optionally be included to test the converse condition.

Alias condition examples:

```
LEAD_CHARACTER IS A DELIMITER
LAST_DIGIT IS NOT A TERMINATOR
```

12.2.1.7 EVENT Condition

The process specification provides for a WAIT UNTIL statement whereby the process is stopped and held at that statement depending on the value of the binary event attribute (it may be 0 or 1). It automatically continues when the event attribute is changed. The format of the event condition is shown below.

$$\text{event_name} \quad [\text{EVENT}] \quad \text{IS} \quad [\text{NOT}] \quad \left\{ \begin{array}{l} \text{event_value} \\ \text{alias_name} \end{array} \right\}$$

An example of the use of the EVENT attribute along with the ALIAS is shown below:

```
1 MY_NEXT_EVENT          EVENT
    ALIAS GO              VALUE 1
    ALIAS STOP            VALUE 0

IF MY_NEXT_EVENT IS STOP
    WAIT UNTIL MY_NEXT_EVENT IS GO .
```

The event_value is stored as an integer, and may NOT be changed by a MOVE or arithmetic statement. Group moves must preserve the value assigned by a SET statement.

12.2.1.8 RELATION Condition

A relation condition causes a comparison of an attribute with either another attribute, a literal, or a named constant.

$$\left\{ \begin{array}{l} \text{attribute_name_1} \\ \text{arithmetic_construct} \end{array} \right\} \quad \left[\begin{array}{l} \text{IS} \\ \text{ARE} \end{array} \right] \quad [\text{NOT}] \quad \left\{ \begin{array}{l} \text{GREATER} \text{ [THAN]} \\ \text{LESS} \text{ [THAN]} \\ \text{EQUAL} \text{ [S] [TO]} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{attribute_name_2} \\ \text{literal} \\ \text{named_constant} \\ \text{arithmetic_construct} \end{array} \right\}$$

The clarity words IS, ARE, TO, and THAN are optional but may be added to improve readability. The keyword NOT may optionally be included to test converse conditions.

When comparing numeric attributes (i.e., REAL, DREAL, INTEGER, INDEX, INDEX_1, or DECIMAL) the numeric values are compared.

When evaluating a non-numeric relation condition, the following collating sequence is used to determine the relative value of VisiSoft characters, in ascending order from the space character to the underline character as defined in the VisiSoft Collating Sequence below.

VisiSoft *Conditional Language* Collating Sequence

Character	Meaning
	blank (space)
'	single quote/apostrophe
(left parenthesis
)	right parenthesis
*	asterisk (multiplication sign)
+	plus sign
,	comma
-	minus sign
.	period (decimal point)
/	slash (division sign)
0,1,...,9	digit
=	equal sign
A,B,...,Z	uppercase letter
_	underline

Relation *condition* examples

```

TRANSCIVER_NUM IS GREATER THAN TOTAL_TRANSCIVERS
CLOCK_TIME IS NOT GREATER THAN 25
SIN(THETA_1 + THETA_2) EQUAL ZERO

```

12.2.1.9 SIGN Condition

The sign condition determines whether or not the value of a numeric attribute (i.e., an attribute described with an INTEGER, REAL, DREAL, INDEX, or INDEX_1 clause) is NEGATIVE (less than zero), POSITIVE (greater than zero), or ZERO (equal to zero).

$$\left\{ \begin{array}{l} \text{attribute_name} \\ \text{arithmetic_construct} \end{array} \right\} \left[\begin{array}{l} \text{IS} \\ \text{ARE} \end{array} \right] [\text{NOT}] \left\{ \begin{array}{l} \text{POSITIVE} \\ \text{NEGATIVE} \\ \text{ZERO} \end{array} \right\}$$

12.2.1.10 FILE Condition

The file condition can be used to determine whether a file already EXISTS before creating new ones. It can also determine whether an existing file is EMPTY. The format is:

$$\left\{ \begin{array}{l} \text{attribute_name} \\ \text{nonnumeric_literal} \end{array} \right\} \left\{ \begin{array}{l} \text{EXISTS} \\ [\text{DOES}] \text{NOT EXISTS} \\ [\text{IS}] [\text{NOT}] \text{EMPTY} \end{array} \right\}$$

The attribute_name must contain the filename, and the non numeric literal is of the form 'file_name', either of which can be a fully qualified path_name/file_name of up to 40 characters.

File condition example

```
IF MY_EXTERNAL_FILE EXISTS
AND MY_EXTERNAL_FILE IS NOT EMPTY
    ASSIGN MY_EXTERNAL_FILE TO MY_EXTERNAL_RESOURCE.
```

12.2.1.11 COMPOUND Conditions

Two or more simple conditions can be combined to form a compound condition. Parentheses may be used to group simple conditions together. Each subsequent simple condition in the compound condition must begin with one of the logical operators AND or OR.

Logical Operator	Meaning
OR	Logical inclusive OR, i.e., either or both are true
AND	Logical conjunction, i.e., both are true
NOT	Logical negation

Table 12-2 shows the relationships between the logical operators and simple conditions A and B.

Table 12-2 Logical Operators and the Resulting Values Upon Evaluation

A	B	A AND B	A OR B	NOT A	NOT (A AND B)	NOT A AND B	NOT (A OR B)	NOT A OR B
True	True	True	True	False	False	False	False	True
False	True	False	True	True	True	True	False	True
True	False	False	True	False	True	False	False	False
False	False	False	False	True	True	False	True	True

Logical evaluation of compound conditions begins with the least inclusive (innermost) pair of parentheses and proceeds to the most inclusive (outermost). If the order of evaluation is not specified by parentheses, the expression is evaluated in the following order:

- (1) Arithmetic constructs (in the sequence **, * and /, + and -)
- (2) Relational operators
- (3) NOT condition
- (4) AND and its surrounding conditions are evaluated first, starting at the left of the expression and proceeding to the right.
- (5) OR and its surrounding conditions are then evaluated, also proceeding from left to right.

Each simple condition within a compound condition must be written fully.

For example:

```
A EQUALS B OR C OR D
```

is not valid, and must be written as

```
A EQUALS B
OR A EQUALS C
OR A EQUALS D
```

Compound condition examples

```
TOTAL IS GREATER THAN ZERO
OR TOTAL IS EQUAL TO ZERO
```

```
CLOCK_TIME IS GREATER THAN 25
AND (TELEPHONE STATUS IS BUSY
OR START_INDEX IS POSITIVE)
```

12.2.1.12 RUN_TYPE Condition

The user can test to determine if a normal task is being run or if an optimization task has been invoked by testing the VisiSoft RUN_TYPE attribute.

$$\text{RUN_TYPE IS [NOT] } \left\{ \begin{array}{c} \text{RUN} \\ \text{OPTIMIZATION} \end{array} \right\}$$

Example:

```
IF RUN_TYPE IS OPTIMIZATION
AND SOLUTION_TYPE IS OPTIMAL
PRINT 'OPTIMAL SOLUTION FOUND'.
```

12.2.1.13 OPTIMAL/FEASIBLE SOLUTION Condition

When running optimization, the user can test to determine if a feasible or optimal solution has been found by testing the SOLUTION_TYPE attribute.

$$\text{SOLUTION_TYPE IS [NOT] } \left\{ \begin{array}{c} \text{FEASIBLE} \\ \text{OPTIMAL} \end{array} \right\}$$

See the example in 12.2.1.12 above.

12.2.2 THEN AND ELSE CLAUSES

VisiSoft IF statements are designed to read like English, and must end with a period. The only exception to this rule arises when IF statements are used in a CASE type statement (see Section 12.2.4), in which case the CASE statement must end with a period. Most importantly, the rule hierarchies eliminate the need for nested IFs, i.e., and IF statement inside of an IF statement. We note that CASE statements are not nested IFs.

Referring to the format diagram in Section 12.2, the *THEN clause* consists of the optional word THEN and the words and constructs following it, up to the ELSE keyword or period. The optional *ELSE clause* consists of the ELSE keyword up to the end of the IF statement.

When an IF statement is executed, the following action is taken:

If the condition is true, the THEN clause is executed. Control is then passed to the statement following the period denoting the end of the IF statement.

- If the THEN clause contains the keywords NEXT STATEMENT, no other constructs are permitted, and control passes directly to the statement following the period.
- If the THEN clause contains the keywords EXIT [THIS] RULE, the rule is exited immediately, and control passes back to the statement following the one executing the rule.

If the condition is false, the ELSE clause is executed and control passes to the statement following the period.

- If the ELSE option is omitted or if the ELSE clause contains the keywords NEXT STATEMENT, control passes directly to the statement following the period.
- If the THEN clause contains the keywords EXIT [THIS] RULE, the rule is exited immediately, and control passes to the statement following the one executing the rule.

Within the THEN and ELSE clauses, each statement must begin on a separate line.

12.2.3 IF - THEN - ELSE STATEMENTS

Referring again to the format diagram in Section 12.2, one or more of the statements in the THEN or ELSE clauses may themselves be conditional statements. These are called *nested* IF statements. Only the final clause of a nested IF statement must have a period at its end. Internally nested IF statements will be terminated implicitly by either the IF or ELSE keywords or by the period used to terminate the main IF statement of which it is a part.

IF statements contained within IF statements must be considered as paired IF and ELSE combinations, proceeding from left to right. Any ELSE encountered must be considered to apply to the immediately preceding IF that has not already been paired with an ELSE. If any ambiguities may occur, the NEXT STATEMENT option may be used to clarify the pairings.

IF examples

```
IF CATEGORY IS LOCAL
    INCREMENT TOTAL_LOCAL_CALLS.

IF OUTGOING_LINE IS NOT BUSY
    THEN EXECUTE CONNECT_CALL.

IF OUTGOING_LINE IS NOT BUSY
    SCHEDULE CONNECT_CALL
ELSE INCREMENT CALLS_BLOCKED.

IF OUTGOING_LINE IS NOT BUSY
AND LINKS_IN_USE ARE LESS THAN 100
    INCREMENT LINKS_IN_USE
    SCHEDULE CONNECT_CALL.

IF CLOCK_TIME IS POSITIVE
AND RADIO IS READY
    EXECUTE MESSAGES_SENT
ELSE SCHEDULE TRANSMIT IN 3 MINUTES.

IF TOTAL_ORDERS_COMPLETED ARE GREATER THAN 100
    THEN EXIT THIS RULE
ELSE EXECUTE MANUFACTURE_GOODS
    SCHEDULE STOCK_UP IN 1 HOUR.
```

12.2.4 IF - THEN - ELSE CASE STATEMENT

A special form of a IF statement is called the IF - THEN - ELSE CASE statement, and takes the form:

```
IF condition_1
    statement_1
ELSE IF condition_2
    statement_2
.
.
.
ELSE IF condition_n
    statement_n.
```

This is a convenient structure to use when a small number of different conditions are to be tested, each followed by a single construct. The keyword THEN may optionally be included before the statement. Only the statement following the first satisfied condition will be executed.

CASE examples

```
IF RADIO IS READY
    RESUME TRANSMIT NOW
ELSE IF RECEIVER IS TRANSMITTING
    RESUME TRANSMIT IN 20 SECONDS
ELSE IF TRANSMITTER IS NOT ON
    EXECUTE START_TERMINAL.
```

12.2.5 EXECUTE RULE_POINTER STATEMENT

This statement may be used to gain substantial speed when a large number of IF - THEN - ELSE CASE type statements occurs, or when the decision mechanism for setting the condition statement is complex. In the later case, the decisions regarding which rules to execute may be spread over multiple rules themselves. When one gets to the IF - THEN - ELSE statement, one cannot repeat the logic. In this case, the EXECUTE statement defined below can use the RULE_POINTER clause shown below which provides a great simplification, and can be used without an IF - THEN - ELSE statement since the decision has already been resolved by selecting the desired RULE using the RULE_POINTER.

EXECUTE rule_pointer RULE.

This is a most convenient structure to use when a large number of different conditions are to be tested, each with corresponding rules to branch to. Up to 50 different rule names can be invoked by the rule_pointer. The rule name assigned to the rule_pointer by the SET RULE statement (refer to Section 12.1.7) will be executed when this statement is encountered.

EXECUTE RULE examples

```
EXECUTE CURRENT_SECTION RULE
EXECUTE EXISTING_CASE RULE
```

where the rule names assigned to CURRENT_SECTION and EXISTING_CASE by the SET RULE statement will be the rules that get executed.

12.2.6 CALL PROCESS_POINTER STATEMENT

For reasons similar to the RULE_POINTER, this is a much faster and more convenient structure to use when a large number of different conditions are to be tested, each with corresponding processes to branch to, or when decisions regarding which process to CALL may be spread over multiple rules. The CALL PROCESS_POINTER statement is similar to the EXECUTE RULE_POINTER statement above, having the form:

CALL process_pointer PROCESS.

Up to 4000 different process names can be invoked by the process_pointer. The process name is assigned to the process_pointer by the SET PROCESS statement (refer to Section 12.1.8) will be called when this statement is encountered.

CALL PROCESS_POINTER examples

```
CALL CURRENT_PROCESS PROCESS
CALL NEXT_PROCESS PROCESS
```

where the process names assigned to CURRENT_PROCESS and NEXT_PROCESS by the SET PROCESS statement will be the processes that get called. Note that the process being called cannot be in an external library.

12.3 PROCESS LEVEL CONTROL STATEMENTS

There are three keywords which deal with flow of control at the process level; these are EXECUTE, SEARCH, and CALL. Each is now described in turn.

12.3.1 EXECUTE STATEMENT

The EXECUTE statement is used to depart from the normal instruction sequence within a rule in order to execute another rule a specified number of times, or until a predetermined condition is satisfied. This statement is defined within three different formats.

Format-1

$$\text{EXECUTE } \left\{ \begin{array}{l} \text{rule_name} \\ \text{rule_pointer RULE} \end{array} \right\} \left[\begin{array}{l} \left\{ \text{unsigned_numeric_literal} \right\} \text{ TIMES} \\ \left\{ \text{attribute_name} \right\} \end{array} \right]$$

Format-2

$$\text{EXECUTE } \left\{ \begin{array}{l} \text{rule_name} \\ \text{rule_pointer RULE} \end{array} \right\} \text{ UNTIL condition}$$

Format-3

$$\begin{aligned} &\text{EXECUTE } \left\{ \begin{array}{l} \text{rule_name} \\ \text{rule_pointer RULE} \end{array} \right\} \left\{ \begin{array}{l} \text{INCREMENTING} \\ \text{DECREMENTING} \end{array} \right\} \text{ attribute_name_1} \\ &\left[\text{FROM } \left\{ \begin{array}{l} \text{numeric_literal} \\ \text{attribute_name_2} \end{array} \right\} \right] \left[\text{BY } \left\{ \begin{array}{l} \text{unsigned_numeric_literal} \\ \text{attribute_name_3} \end{array} \right\} \right] \\ &\left[\begin{array}{l} \text{TO } \left\{ \begin{array}{l} \text{numeric_literal} \\ \text{attribute_name_4} \end{array} \right\} \\ \text{UNTIL condition} \end{array} \right] \end{aligned}$$

Each attribute name and numeric literal must represent integer or index values. The condition may be any one of the thirteen condition types described in Section 12.2.1.

Whenever an EXECUTE statement is executed, control is transferred to the first statement of the rule named *rule name*. Control is always returned to the statement immediately following the EXECUTE statement.

Rules For Format-1

The following rules apply to the use of a Format-1 EXECUTE statement:

- (1) If the TIMES option is omitted, the named rule is executed once.
- (2) If the value of attribute_name is zero or negative at the time the EXECUTE statement is initiated, control passes to the statement following the EXECUTE statement, i.e. the TIMES option is tested first.
- (3) Once the EXECUTE statement has been initiated, any change in value of attribute_name has no effect in varying the number of times the rule is initiated.
- (4) A statement in a rule cannot execute the rule currently in process.
- (5) Executing rules recursively can produce erroneous results.

Examples

```
EXECUTE NEXT_CALL  
EXECUTE NEXT_CALL 5 TIMES  
EXECUTE NEXT_CALL TOTAL_CALL TIMES
```

Rules For Format-2

The following rule applies to the use of a Format-2 EXECUTE statement:

The specified rule is performed until the *condition* specified by the UNTIL clause is true. At this time, control is transferred to the statement following the EXECUTE statement. If the condition is true at the time that the EXECUTE statement is encountered, the specified rule is not executed, i.e. *the condition is tested first*.

Examples

```
EXECUTE NEXT_CALL UNTIL NO_LINES ARE AVAILABLE  
EXECUTE NEXT_CALL UNTIL CALLS_ANSWERED ARE EQUAL TO  
    CALLS_WAITING  
EXECUTE READ_MESSAGE UNTIL LEAD_CHARACTER IS A DELIMITER
```

Rules For Format-3

The following rules apply to the use of a Format-3 EXECUTE statement:

- (1) The initial value of the FROM clause (attribute_name_2 or the default value 1) is always set prior to the first test. The value of attribute_name_1 is always tested *prior to executing the rule* to determine if the TO or UNTIL clause is satisfied. If satisfied, the rule is *not* executed. If not satisfied, the rule is executed. Immediately after the rule is executed, attribute_name_1 is incremented (or decremented).
- (2) If the FROM or BY part of the EXECUTE statement is omitted, the default is FROM 1 or BY 1. If the BY option is used, attribute_3 must be a positive integer. If the TO option is omitted, then the UNTIL clause is required to limit continued execution (TO and UNTIL are mutually exclusive). When the DECREMENTING option is used, the use of FROM is mandatory.
- (3) While rule_name is being executed, changing the value of attribute_name_1 (in the INCREMENTING / DECREMENTING clause) or attribute_name_3 (in the BY option) or attribute_name_4 (in the TO option) can change the number of times the rule is executed. Changes in value of attribute_name_2 (in the FROM option) will have no effect in altering the number of times the rule is to be executed, after execution starts.

Examples

```
EXECUTE NEXT_CALL INCREMENTING CALL_NUMBER BY 1 FROM 1 TO 99
EXECUTE ANSWER_CALL INCREMENTING CALLS_ANSWERED
      UNTIL CALLS_ANSWERED ARE GREATER THAN CALLS_WAITING
```

Note that, in this example, if the attribute CALLS_ANSWERED is changed during execution of the rule ANSWER_CALL, the number of times the rule is executed may be affected.

12.3.2 SEARCH TABLE STATEMENT

Designed for speed, the SEARCH table statement provides for automatic searching of complex hierarchical tables over all indices, and execution of a designated rule when the specified table conditions are found to be true. The form of the SEARCH table statement is as follows:

```
SEARCH table_name
OVER attribute_name_1  [search_range_1]
  [AND attribute_name_2  [search_range_2] ]
    [AND attribute_name_3  [search_range_3] ... [search_range_6] ]
      {EXECUTING rule_name  [WHEN condition_1] [UNTIL condition_2]}
        {
          UNTIL condition_3
        }
```

where search_range_n is of the form:

```
[FROM {unsigned_numeric_literal}] [BY {numeric_literal}] [TO {unsigned_numeric_literal}]
[attribute_name_5] [attribute_name_4] [attribute_name_6]
```

The conditions take the standard form of a VisiSoft condition. Table_name is the name of the attribute containing the most interior QUANTITY clause to be searched. When the search_range is not specified, the implied search_range will be over the full QUANTITY of the table, using the attribute_name specified as the variable index.

The implied search range starts FROM 1 and runs TO the value in the QUANTITY clause. Then attribute_name_1, attribute_name_2, and attribute_name_3 will be incremented by one in all cases.

If the BY option is used, attribute_name_4 can be a positive or negative integer. In the FROM and TO options, attribute_name_5, attribute_name_6, and the unsigned numeric literal must contain positive integers if used. If attribute_name_4 is negative, then attribute_name_5 in the FROM option must be greater than one, being the starting point of the count down.

If the TO option is used when attribute_name_4 is negative, then attribute_name_5 must be greater than attribute_name_6 for rule_name to be executed.

When multiple search_ranges are used, they will be varied by holding the first named attribute in the OVER clause to its FROM value, while varying the last named attribute through its entire search range. For example, in the case of an attribute with 3 subscripts, each of quantity 2, the search order would be:

```
X(1,1,1), X(1,1,2),
X(1,2,1), X(1,2,2),
X(2,1,1), X(2,1,2),
X(2,2,1), X(2,2,2).
```

As an example, consider the following attribute structure:

NUMBER_OF_TRANSCEIVERS	INDEX
RECEIVER	INDEX
TRANSMITTER	INDEX
LINK_CONNECTIVITY_VECTOR	QUANTITY(500)
1 CONNECTIVITY_MATRIX	QUANTITY(500)
2 PROPAGATION_LOSS	REAL
2 SIGNAL_TO_NOISE_RATIO	REAL
2 LINK	STATUS GOOD
	FAIR
	POOR

To SEARCH this two-dimensional table executing TRANSMISSION for every LINK that is GOOD, one can use the following statement:

```
SEARCH CONNECTIVITY_MATRIX OVER RECEIVER, AND TRANSMITTER
EXECUTING TRANSMISSION
    WHEN LINK(RECEIVER, TRANSMITTER) IS GOOD
```

To limit the search range to NUMBER_OF_TRANSCEIVERS instead of covering the 500 by 500 range, one would write the following:

```
SEARCH CONNECTIVITY_MATRIX
OVER RECEIVER TO NUMBER_OF_TRANSCEIVERS
AND TRANSMITTER TO NUMBER_OF_TRANSCEIVERS
EXECUTING TRANSMISSION
    WHEN LINK(RECEIVER, TRANSMITTER) IS GOOD
```

To search the LINK_CONNECTIVITY_VECTOR to find the good links to a particular RECEIVER over the same range of TRANSMITTERs, one would write the following:

```
RECEIVER = SELECTED_RADIO
SEARCH LINK_CONNECTIVITY_VECTOR
OVER TRANSMITTER TO NUMBER_OF_TRANSCEIVERS
EXECUTING TRANSMISSION
    WHEN LINK(RECEIVER, TRANSMITTER) IS GOOD
```

12.3.3 CALL STATEMENT

The CALL statement causes control to be transferred directly from one process to another. This can be done using a process_pointer PROCESS as defined in Section 12.2.6. When using Independent (IND) modules, e.g., when running on parallel processors, the CALLED process must be within the same IND module.

$$\text{CALL } \left\{ \begin{array}{l} \text{IP_process_name} \\ \text{IP_process_pointer IP_PROCESS} \end{array} \right\}$$
$$\left[\begin{array}{l} [\text{WITH}] \text{ INSTANCE } \left\{ \begin{array}{l} \text{num_literal_1} \\ \text{attribute_name_1} \end{array} \right\} \left[\dots \left\{ \begin{array}{l} \text{numeric_literal_6} \\ \text{attribute_name_6} \end{array} \right\} \right] \\ \text{USING } \left\{ \begin{array}{l} \text{num_literal_1} \\ \text{attribute_name_1} \end{array} \right\} \left[\dots \left\{ \begin{array}{l} \text{numeric_literal_6} \\ \text{attribute_name_6} \end{array} \right\} \right] \end{array} \right]$$

The execution of a CALL statement causes control to pass immediately to the process whose name is specified. When the called process is completed, control immediately returns to the statement following the CALL statement in the calling process. Called processes may also contain CALL statements. However, a "called" process must not contain a CALL statement that directly or indirectly calls itself, or any originator of a call chain to that process. Recursion is neither necessary nor supported (it is slow). A process may CALL, SCHEDULE, and CANCEL up to 30 different processes. Up to 50 CALL, SCHEDULE, and CANCEL statements may appear in one process. It may be CALLED, SCHEDULED, or CANCELED by up to 100 different processes that reside within the same process directory.

12.3.3.1 The INSTANCE Option

Section 11.3 explained the passing of INSTANCE and other pointers to processes within instanced or non-instanced modules. If the process being called is within an INSTANCED module, then the number of instance pointers must match the number of instance layers to the elementary module containing the called process (up to a maximum of 6), *even when the calling process resides within the same instanced module*. When the CALL statement is processed, the values of the instance pointers are used to identify the hierarchy of instances leading to the elementary module containing the desired process to be executed. These instance pointers also specify the instances of each resource within the hierarchy of instanced modules containing the process to be executed.

The instance pointers listed after the keyword INSTANCE will be matched with the instances listed for the process being called in accordance with the order of listing. This order must match that of the hierarchy of instanced modules from top to bottom. It can be viewed using the QUERY option. Any numeric literals listed in an INSTANCE clause must be positive integers. Attributes listed in an INSTANCE clause must be INDEX_1, INDEX, or INTEGER type, and must not be subscripted.

The other pointers may be used with or without an INSTANCE pointer. Examples are illustrated below. Note that the qualifying name INSTANCE must be used. In the case of other pointers, the USING qualifier must be used.

CALL INSTANCE examples

```
CALL NEXT_BROADCAST  
  
CALL ATTENUATION_CALC  
    WITH INSTANCE SOURCE, DESTINATION
```

CALL USING examples

```
CALL ATTENUATION_CALC  
    USING PARM_1, PARM_2  
  
CALL ATTENUATION_CALC  
    WITH INSTANCE SOURCE, DESTINATION  
    USING PARM_1, PARM_2
```

12.3.3.2 Calling Processes by Generic Names

When changing a process to modify its functions, one may want to keep a copy of the old process, giving the new process a new name. To avoid changing the call statements in those processes that call the old process, one can use the old process name as the new process *generic name*. Although GSS process names must be unique, generic names can be the same as process names, and can be reused by more than one process. Thus, the use of generic names requires resolution of referenced names, since the system must know whether the call statement applies to the old process or the new process whose generic name is the same as the old process name.

```
CALL NEXT_BROADCAST
```

Resolution of generic names is accomplished as follows. A process invoked by its generic name in a call statement must exist in a model that is named in the MODEL SECTIONs of a Simulation Control Specification, reference Figure 13-1. This model could be either hierarchical or elementary. Figure 12-2 illustrates the use and resolution of generic names by decomposition of the system into models. All the process and generic names shown in Figure 12-2 (a) are listed below.

PROCESS NAME	GENERIC NAME
P_A	
P_B	
P_C	
P_D	
P_N	P_K

In reviewing Figure 12-2 (b), old process P_K is called from both processes P_A and P_D. However, the designer actually wants to call the new process P_N, using the generic name P_K. This is accomplished by naming the desired process to be called inside a model named in the MODEL section of a Task Control Specification, or in the DEFINITION, IDENTIFICATION, MODEL, or EVALUATION SECTIONs of a Simulation Control Specification. Only the *process names* (as opposed to the *generic names*) can be designated in a model, and these are unique. If two processes within a model use the same generic name, CALLs by that generic name cannot be resolved.

GENERIC NAME REFERENCES

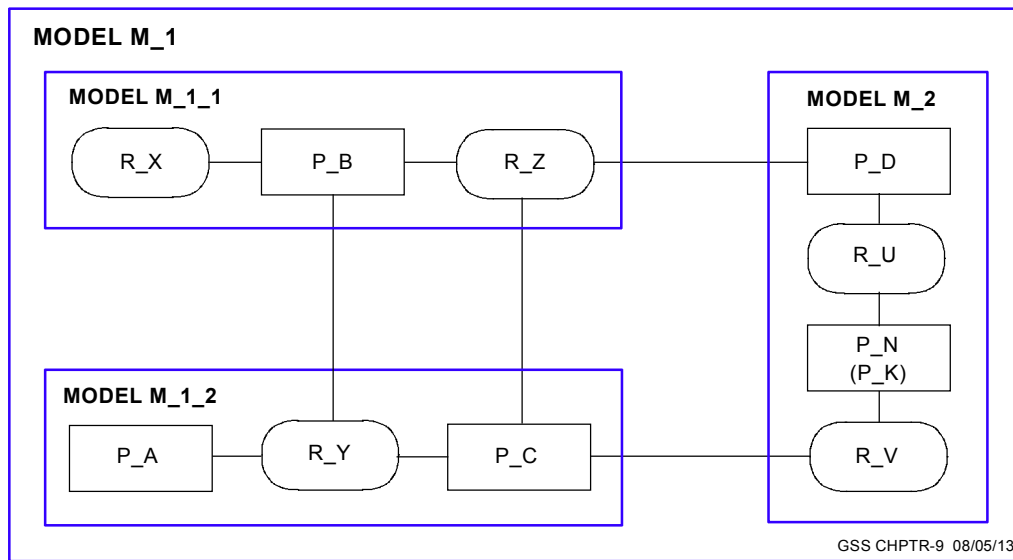


Figure 12-2 (a). Use of module boundaries to resolve generic name references.

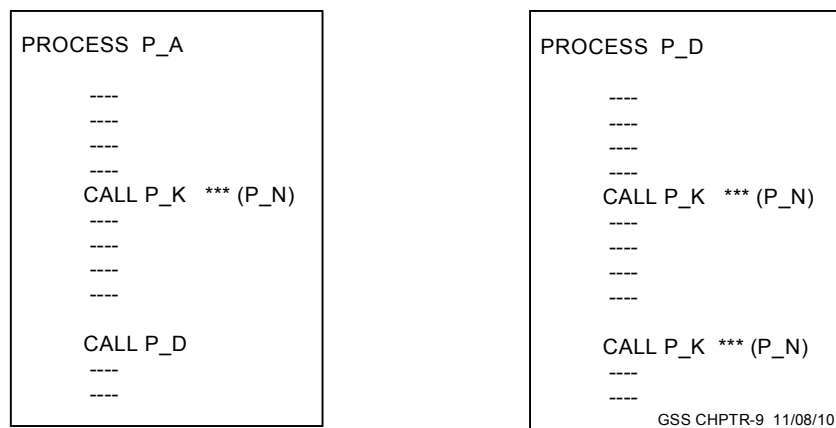


Figure 12-2 (b). Illustration of generic name call references.

12.3.3.3 Library CALL Statement

When calling a process contained in an external GSS library, a *library CALL* statement must be used to identify the particular process, module, and library selected. In general, no source code will be available for that process, being stored in object form in the corresponding `library_name.a` file. This CALL statement identifies the interface resources shared by the calling process that corresponds to the aliased resources in the called module. The order of resources in the USING list must be the *same as that in the called module* - alphabetical order by name. The syntax is as follows:

```
CALL process_name IN module_name [IN] library_name
    USING resource_1 [ , ..., resource_m]
```

The `process_name` must be that of a process in the specified library module. Because of the way GSS library modules are prepared, every process name in a module is unique, and every module name in a library is unique. The names of library files in a user's directory must also be unique. Every process name in a library is automatically appended with the `module_name` as well as the `library_name` when it is prepared, and must be addressed as such when called from the outside. This guarantees uniqueness even though the same `process_name` may be used in different library modules. There is no limitation on CALLs to processes that reside within library modules. The USING clause is used to identify resources SHARED at the interface to a library, corresponding to the aliased resource(s) attached to the called library process.

Library calls may be chained by calling one library from another library. When calling a library that uses another library, it is up to the user to insure that all modules in a chain are prepared with the desired versions.

EXAMPLES:

```
CALL COVARIANCE IN KALMAN FILTERS USING A_MATRIX
CALL GET_EIGENVALUES IN INVERT MATRIX USING F_MATRIX, R_MATRIX
```

In the above examples, COVARIANCE is a process in model KALMAN that resides inside library FILTERS. The A_MATRIX resource is *shared* by the calling routine and corresponds to an *aliased* resource attached to COVARIANCE.

GET_EIGENVALUES is a process in module INVERT that resides within library MATRIX. The F_MATRIX and R_MATRIX resources are *shared* by the calling process and correspond to *aliased* resources attached to GET_EIGENVALUES. The order of the resources in the USING clause is determined by the order of the aliased resources in the COVARIANCE process. *Aliased resources are always attached in alphabetical order.*

When using parallel processors, library modules will be copied onto each processor containing modules that call that library module.

12.3.4 SCHEDULE STATEMENT

The SCHEDULE statement is used to schedule another process, or itself, to run either immediately after the process that schedules it, or at a specified time in the future. When a process is scheduled, it is placed in the VisiSoft schedule to be run at the specified (relative) time in the future. When used on a parallel processor, the SCHEDULE statement starts a thread within an IND Module that runs to completion on the specified processor. Multiple processes may be CALLED by processes in the thread. The thread terminates when the statements in the scheduled process terminate. The SCHEDULE statement is shown below.

$$\begin{aligned} &\text{SCHEDULE } \left\{ \begin{array}{l} \text{process_name} \\ \text{process_pointer PROCESS} \end{array} \right\} \quad [\text{IN IND_module_name}] \\ &\left[\begin{array}{l} [\text{WITH}] \text{ INSTANCE } \left\{ \begin{array}{l} \text{numeric_literal_1} \\ \text{attribute_name_1} \end{array} \right\} \left[\dots \left\{ \begin{array}{l} \text{numeric_literal_N} \\ \text{attribute_name_N} \end{array} \right\} \right] \end{array} \right] \\ &\left[\begin{array}{l} \text{USING } \left\{ \begin{array}{l} \text{numeric_literal_N+1} \\ \text{attribute_name_N+1} \end{array} \right\} \left[\dots \left\{ \begin{array}{l} \text{numeric_literal_6} \\ \text{attribute_name_6} \end{array} \right\} \right] \end{array} \right] \\ &\left[\begin{array}{l} \left\{ \begin{array}{l} \text{AT} \\ \text{IN} \end{array} \right\} \left\{ \begin{array}{l} \text{numeric_literal} \\ \text{attribute_name} \end{array} \right\} [\text{time_units}] \\ \text{NOW} \end{array} \right] \quad [[\text{WITH}] \text{ PRIORITY priority_code}] \end{aligned}$$

The VisiSoft SCHEDULE statement serves two purposes. One is generally concerned about simulated clock times. In this case, clock time is generally relative, being used to synchronize the unfolding of simulated physical events inside models. In the case of nonlinear models, one must reconcile the nonlinear behavior before proceeding to the next time step. This requires subject area experts to determine what is required to obtain the level of model accuracy that ensure validity of the simulated results. Simulations may also be embedded in real-time control systems wherein they are really software systems that must be tied to the real-time clock.

Examples of real-time software are real-time control systems, transaction processing systems, or database update systems. In all of these cases, the clock may be the real-time clock, or a relative version of the real-time clock so that events are processed in a desired time sequence dictated by the application. Time is always relative to the accuracy of the clock differences. Thus it is up to the application system expert to determine how far the software clock can drift from a real-time clock.

Using VisiSoft, both simulation and software applications may run on a single processor or a parallel processor. If an application has a number of IND modules that are designed to run on a parallel processor, then it can also run on any number of processors except that the number of processors used will not exceed the number of IND modules. If the number of processors is less than the number of IND modules, then modules will be stacked to fit the number of processors. There is no need to change the application architecture or code to run on either. The only change required is in the Control Specification which determines the number of processors to be used, and how the IND Modules may be initialized across processors.

12.3.4.1 PROCESS NAME

The `process_name` must be that of a process specified within the Simulation Control Specification. This may be done through a model/module name, or a library name. This can also be done using a `process_pointer` PROCESS as defined in Section 12.2.6.

Example

```
SCHEDULE CLOSE_DOWN AT STOP_TIME
```

12.3.4.2 IND MODULE NAME

When using parallel processors, the SCHEDULE statement is used to start threads in different IND Modules as well as the same IND Modules, where IND Modules may reside on different processors and well as the current processor. When using parallel processors, the IND Module name must be specified. This implies that the IND Module must be designated as such in the Architecture Environment. In the following example, PLATFORM_MODULE is an IND Module.

Example

```
SCHEDULE START_PLATFORMS IN PLATFORM_MODULE NOW
```

12.3.4.3 INSTANCE & INDEX POINTERS

INSTANCE POINTERS and *other* POINTERS are described in Section 4.2. They are used to identify both hierarchical module instances and other indices within a resource. If a process in a particular Module Instance is to be scheduled, the module instance must be specified by listing the pointer - or set of pointers - that identifies it, in order from the highest level of hierarchy to lowest, after the keyword INSTANCE - *explicitly* - even when scheduling itself.

If the process being scheduled also uses *other* POINTERS then a matching number of other pointers must be specified. The total number of pointers - INSTANCE plus other - cannot exceed 6.

When the schedule statement is processed, the values of the instance and index pointers are saved to be made available to the scheduled process at the time it executes. In the case of an instanced module, these instance pointers specify the resources within a potential hierarchy of modules containing the process to be executed. In the case of index pointers, they specify the element within a (potential hierarchy of) QUANTITY clause(s) in a resource.

The instance pointers listed after the keyword INSTANCE will be matched with the instances listed for the process being scheduled in accordance with the order of listing. This order may be viewed and changed using the MODIFY PROCESS option, and viewed using the QUERY option. Any numeric literals listed in an INSTANCE clause must be positive integers. However, literals can be negative in assignment statements. Attributes listed in an INSTANCE or INDEX clause must be type INDEX_1, INDEX, or INTEGER, and must not be subscripted.

Example

```
SCHEDULE TRANSMISSION WITH INSTANCE TRANSMITTER  
IN UNIFORM(0.5, 1.5) SECONDS
```

12.3.4.4 AT, IN, NOW

If a process is scheduled with none of the optional AT, IN, or, NOW clauses, it will be scheduled at the current time (same time as the executing process containing the SCHEDULE statement). It can run when the current process completes execution without advancing the schedule clock. Whether or not it runs immediately after the current process will depend upon the PRIORITY of processes scheduled at the same time. SCHEDULE process-name differs from a CALL statement in that control is not passed directly to the scheduled process.

Parallel Processor Considerations

When scheduling a process in an IND module that resides on a different processor, the different scheduler clocks will generally be out of synchronization. When using the IN option, it is up to the designer to determine whether a schedule that falls near the boundary of a ΔT_{\max} interval should fall within or beyond that boundary. This is easily solved by computing an AT time, based upon the application requirements, to be passed to the other processor.

Example

```
SCHEDULE ARRIVAL NOW  
SCHEDULE FLY_BLUE_AIRCRAFT IN IND_MODULE-AIRCRAFT AT END_OF_INTERVAL
```

When the currently executing process contains more than one SCHEDULE NOW statement, the processes it schedules will all begin after the current process terminates. The order in which they are executed, however, will be unpredictable unless priority codes have been assigned (See Section 12.3.4.4). A process may CALL, SCHEDULE, and CANCEL up to 30 different processes. Up to 50 CALL, SCHEDULE, and CANCEL statements may appear in one process. A process may be CALLED, SCHEDULED, or CANCELED by up to 100 different processes that reside within the same process directory.

If the named process is not to be executed at the current time, i.e., the same time at which it is scheduled, the time at which the named process is to be executed must be indicated by one of the keywords AT or IN.

It is possible to specify an absolute time at which execution of the SCHEDULED process is to begin. This must be done relative to the simulation CLOCK_TIME and the numeric literal or attribute value should correspond to a value, which CLOCK_TIME is expected to reach during the simulation run. Time-units may optionally be specified here (refer to Appendix 3). The specified attribute-name must not be subscripted.

Example

```
SCHEDULE CLOSE_DOWN AT STOP_TIME
```

The second option is to specify the time interval which must elapse before the named process is to begin execution. The elapsed time may be either:

- (1) Deterministic - represented by a positive real numeric literal or unsubscripted resource attribute value, or
- (2) Probabilistic - represented by a statistical distribution with specified parameters. The possible distributions are described in Appendix 4.

Example

```
SCHEDULE ARRIVAL IN EXPON(0.5)
SCHEDULE ARRIVAL IN NORMAL(MEAN, VAR)
```

In addition to specifying the elapsed time, units of time may optionally be specified, refer to Appendix 3. The specified attribute-name must not be subscripted.

Examples

```
SCHEDULE CLOSE_DOWN IN 10 MILLISECONDS
SCHEDULE FLY_BLUE_AIRCRAFT IN 1 MINUTE
```

12.3.4.5 WITH PRIORITY

An option in a SCHEDULE statement is to specify a priority code for the process (see the format in Section 12.3.4). This may be used along with one of the keywords AT or IN. A priority-code is a one-digit or two-digit number greater than zero chosen by the user. The priority-code may be a numeric literal or a non-subscripted resource attribute defined as integer. If a resource attribute is used, the attribute value must be within the proper range.

Examples

```
SCHEDULE ARRIVAL IN EXPON(0.5) WITH PRIORITY PRIORITY_VALUE
SCHEDULE ARRIVAL IN EXPON(0.5) WITH PRIORITY 10
```

In the event that more than one process is scheduled to execute at the same time, the one with highest priority (lowest number) will be executed first. If no priority code is specified, GSS assumes a priority of 50. If two processes are scheduled to execute at the same time, with the same priority, their order of execution is unpredictable.

12.3.4.6 SCHEDULING By Generic Names

Processes can be scheduled by their generic names. Section 12.3.3.2 described calling a process by its generic name. All of the discussions in that section apply to scheduling a process by its generic name.

12.3.5 CANCEL STATEMENT

The CANCEL statement removes processes from the SCHEDULE, implying that they are still waiting to be run.

```
CANCEL [ALL] process_name [IN IND_module_name]
```

$$\left[\text{INSTANCE} \begin{array}{c} * \\ \left\{ \begin{array}{l} \text{numeric_literal_1} \\ \text{attribute_name_1} \end{array} \right\} \end{array} \right] \left[\dots \begin{array}{c} * \\ \left\{ \begin{array}{l} \text{numeric_literal_n} \\ \text{attribute_name_n} \end{array} \right\} \end{array} \right]$$

$$\left[\begin{array}{c} \left\{ \begin{array}{l} \text{AT} \\ \text{IN} \end{array} \right\} \left\{ \begin{array}{l} \text{numeric_literal} \\ \text{attribute_name} \end{array} \right\} \text{ [TIME_UNITS]} \\ \text{NOW} \end{array} \right]$$

The CANCEL statement may be used alone, or one of the phrases AT or IN may be included to indicate when the process is to be cancelled. The ALL option may be used along with either AT or IN. A process may CALL, SCHEDULE, and CANCEL up to 30 different processes. Up to 50 CALL, SCHEDULE, and CANCEL statements may appear in one process. It may be CALLED, SCHEDULED, or CANCELED by up to 100 different processes that reside within the same process directory.

The CANCEL statement causes the first occurrence of the named process to be cancelled on or after the time specified by the AT or IN options if they are used, or immediately after the current process if they are not used. If the ALL option is used, all occurrences currently scheduled following the first one cancelled are also cancelled.

Figure 12-3 illustrates how the CANCEL statement operates. The statement CANCEL TRANSMISSION is executed at time T1, and the time specified in the AT option is T3. GSS will cancel one or more occurrences of the process TRANSMISSION, depending on whether the ALL option is specified. The first occurrence was scheduled to execute at time T2, with subsequent occurrences at times T4 and T5. The first occurrence after the specified CANCEL time, i.e. the one at T4, will be cancelled. IF CANCEL ALL is specified, all subsequent scheduled occurrences of TRANSMISSION will also be cancelled.

Parallel Processor Considerations

Note that the same parallel Processor Considerations apply to the CANCEL statement as applied to the SCHEDULE statement as described in Section 12.3.4.4.

Examples

```
CANCEL NEXT_CALL
```

```
CANCEL ALL NEXT_CALL
```

```
CANCEL FLY_BLUE_AIRCRAFT IN IND_MODULE_AIRCRAFT AT TIME_BEFORE_INTERVAL
```

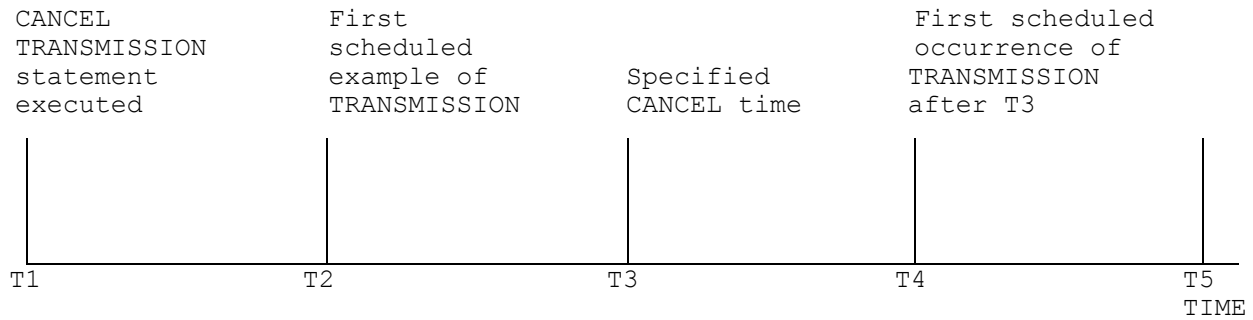


Figure 12-3. Operation of the CANCEL statement.

12.3.5.1 IND MODULE NAME

When using parallel processors, the CANCEL statement is used to cancel threads that are in different IND Modules as well as in the same IND Module, where IND Modules may reside on different processors as well as on the current processor. When using parallel processors, the IND Module name must be specified. This implies that the IND Module must be designated as such in the Architecture Environment. In the following example, PLATFORM_MODULE is an IND Module.

Example

```
SCHEDULE START_PLATFORMS IN PLATFORM_MODULE NOW
```

12.3.5.2 AT, IN

The AT phrase must include a numeric literal or unsubscripted attribute name whose value corresponds to a specific clock time. The AT phrase cancels the first occurrence of the process on or after an absolute time (clock time). The AT phrase in conjunction with the ALL option cancels all occurrences of the process on or after the specified clock time. Time units may optionally be specified here (refer to Appendix 3).

Examples

```
CANCEL DIAL_UP AT 100
CANCEL ALL TRANSMISSION AT STOP_TIME SECONDS
```

The IN phrase also cancels the first occurrence of the process after a specific time interval has elapsed (relative to current time). The elapsed time may be defined using any of the options described in Section 12.3.4.3. The IN phrase in conjunction with the ALL option cancels all occurrences of the process after the specified time interval. Time units may optionally be specified (refer to Appendix 3).

Examples

```
CANCEL ALL NEXT_CALL IN 20 SECONDS
CANCEL DIAL_UP IN WAIT_TIME
```

12.3.5.3 INSTANCE

Section 12.3.4.3 explained the use of instance pointers when scheduling processes. When cancelling a process, which uses instance pointers, the keyword **INSTANCE** may be used. The values of the instance pointers listed after the keyword **INSTANCE** will be matched with those for scheduled versions of the process to determine which occurrence of the process is to be cancelled.

If the keyword **INSTANCE** is omitted, the first occurrence of the named process instance will be cancelled. (All occurrences will be cancelled if the **ALL** option is used.)

Any numeric literals listed in an **INSTANCE** clause must be integers. **Attribute_names** listed in an **INSTANCE** clause must represent **INDEX_1**, **INDEX** or **INTEGER** attributes and must not be subscripted.

The special wildcard symbol '*' may appear in an **INSTANCE** clause to indicate that all values of that pointer will be considered as matches when determining which occurrences of the process should be cancelled.

Examples

```
CANCEL FAULT INSTANCE FAULT_CODE
CANCEL ALL FAULT
CANCEL FAULT INSTANCE *
CANCEL RESTART INSTANCE PART_NUMBER, *
```

12.3.5.4 CANCEL By Generic Names

Processes can be cancelled by their generic names. Section 12.3.3.2 described calling a process by its generic name. All of the discussions in that section apply to cancelling a process by its generic name.

12.4 TASK LEVEL CONTROL STATEMENTS

In a real-time control system, one must be able to control multiple tasks that run on a single processor, allowing them to have access to processor resources by turning them on and off in a synchronized manner, or on a carefully managed *as needed* basis. On a parallel processor, one must be able to have them run concurrently to maximize speed. Again, one must be able to synchronize those processes that may run concurrently. We note that a simulation is a task.

In some cases, tasks may be part of a family that starts with a top level task. In other cases, a task may be part of a different family; this is described in Section 12.4.6.

In either case, to exchange information among tasks, one must share Inter-Task Resources. In the case of a family of tasks, these are Local Inter-Task Resources. Information shared between tasks that are not in the same family must use Global Inter-Task Resources.

In the case of tasks within a family, one must have more control, and VisiSoft provides the following statements:

- START a TASK - This provides for starting and initializing a task so that it is running.
- SUSPEND a TASK - This allows for suspension of a running task. For example, a task may want to suspend itself.
- RESUME a TASK - This provides for running a task that has been started but is suspended.
- TERMINATE a TASK- - This shuts down another task so that it can no longer run.
- STOP - This stops the Task.

There are two ways to start family tasks. One is to create a TASK CONTROL SPECIFICATION as described in Chapter 13. The control specification names the lead-off process for that task. When the control specification runs, it starts with the lead process.

The other way to start a task is by using the START statement inside a process. The following sections describe the statements that support the above task control functions from within a process.

12.4.1 STARTING A CONCURRENT TASK

To use the START command within a process, the designer chooses the point at which he wants to start a subordinate task within the controlling task, and issues a START statement. This statement has the following format:

```
START task_name [[AND]WAIT] [WITH WINDOW [WITH TITLE_REGION title_name]]
```

where task_name includes a fully qualified path name of up to 60 characters total. Specifically, they can be fully qualified path/file names.

If the WAIT option is used, the current task will be suspended until the newly stated task is suspended or terminated. Otherwise, task_name starts running concurrently. If the WINDOW option is used, a window is opened for the newly started task. This statement can appear in any rule in a task.

12.4.2 SUSPENDING A TASK

Once a task is running, it can be suspended, i.e., it would not be actively running, but not terminated. A task can be suspended from within itself by using the SUSPEND command. The format for this command is as follows:

$$\text{SUSPEND THIS TASK} \left[\begin{array}{l} \text{FOR delta_time} \\ \text{UNTIL specified_time} \end{array} \right]$$

where delta_time is a time difference and specified_time is a clock time. If a task is already suspended, this command will only affect the time parameters, taking precedence over prior commands. When a task is suspended without the timer options, it will be suspended until resumed.

The designer must be concerned about the state of the system before and after a task is suspended. Particularly, how is the state of a task preserved when suspended? For example, if it is in the middle of an I/O instruction, e.g., getting input from the mouse or waiting for a tape unit, suspension may be questionable.

Suspending Tasks Using the Time Parameters

Handling suspensions using the time parameters - delta_time and clock_time - will depend upon the accuracy limitations of the operating system being used. When a task is suspended using these time parameters, it can be resumed using the RESUME command, Section 12.4.3, prior to the timer running out. In this case, the designer must determine whether or not to suspend the task again to complete the time-out requirement.

12.4.3 RESUMING A SUSPENDED TASK

A suspended task can be resumed immediately by another task. The statement to do this is defined below.

$$\text{RESUME} \left\{ \begin{array}{l} \text{task_name} \\ \text{task_attribute} \end{array} \right\}$$

12.4.4 TERMINATING A TASK

To terminate a task or a subordinate task from a controlling task, one uses the TERMINATE statement. The format is as follows:

$$\text{TERMINATE } \left\{ \begin{array}{l} \text{THIS TASK} \\ \text{task_name} \end{array} \right\}$$

When a controlling task is terminated using the TERMINATE statement, all subordinate tasks are also terminated.

12.4.5 STOP STATEMENT

The STOP statement allows the user to terminate that section of the simulation containing the STOP statement, either immediately or at a specified point in the future, after which time all scheduled processes will not be run. Refer to Chapter 10 describing the sections of the Simulation Control Specification.

$$\text{STOP } \left[\left\{ \begin{array}{l} \text{AT} \\ \text{IN} \end{array} \right\} \left\{ \begin{array}{l} \text{numeric_literal} \\ \text{attribute_name} \end{array} \right\} \text{ [WITH PRIORITY priority_code]} \right]$$

The keyword STOP may be used alone, or one of the AT or IN phrases may be included to specify when the task is to end. When such a phrase is used, the numeric literal or attribute_name should contain a value, which CLOCK_TIME (simulation clock) is expected to reach during the task. Time units may optionally be specified here (refer to Appendix 3). If the priority is not specified, it will automatically be set to 100 to allow the completion of all previously scheduled processes at that time.

12.4.6 RUNNING A SCRIPT OR TASK

To run a script or task, one uses the RUN statement. The format is as follows:

$$\text{RUN } '[\text{path} \mid \text{environment_variable}] \text{ script_or_task_name}' \mid \text{attribute_name [AND WAIT]}$$

The script_or_task_name is the full name of the OS level script or binary executable task name. path is the relative path or absolute path to the script or task. The environment_variable is the OS level environment_variable affecting the path. The maximum length between the quotes is 256 bytes. If the AND WAIT clause is used, the RUN statement will not return until the script or task completes execution.

With respect to tasks, the major difference between START and RUN is that a task executed by the RUN statement will not be in the same task family as the parent task. This implies that the task executed by the RUN statement must share *global* intertask resources with the task containing the RUN statement and the task control statements that apply to a family (SUSPEND, RESUME, and TERMINATE) are not applicable.

12.5 I/O STATEMENTS

The full complement of Input/Output (I/O) and file handling statements are available in GSS. The possible statement types are outlined below:

	<u>SEQUENTIAL</u>	<u>DIRECT</u>		
ACCEPT	READ	READ	OPEN	DYNAMIC FILE
DISPLAY	WRITE	WRITE	CLOSE	ASSIGNMENT
PRINT				
TRACE				

ACCEPT, DISPLAY, PRINT, and TRACE may be used in any process to read information from a keyboard and write information on a terminal screen or printer file, respectively. Files are sequential or direct, and access data with read and write statements. OPEN and CLOSE statements are used to begin and end, respectively, using a file. If data is formatted in ordered sequences of columns, either alphanumeric or numeric, it can be handled by conforming to the Standard File Interface (SFI) formats, and the user need not write any file access statements. The possible I/O statements are described next.

12.5.1 ACCEPT STATEMENT

The ACCEPT statement enables a process to receive direct input from a terminal keyboard or from special computer functions.

$$\text{ACCEPT attribute_name} \left[\text{FROM} \left\{ \begin{array}{l} \text{DATE} \\ \text{DAY} \\ \text{TIME} \end{array} \right\} \right]$$

If the optional FROM clause is omitted, the process will wait until an entry is received from the keyboard. For this reason, the ACCEPT statement is normally preceded by a DISPLAY statement which puts a prompt or message on the screen to which the keyboard is attached. The value entered at the keyboard will be placed directly in the named resource attribute. The value entered must be compatible with the attribute type (INTEGER, CHARACTER, STATUS, etc.). The maximum size of an attribute in an ACCEPT statement is 80 characters. Optionally, the ACCEPT statement may be used to set a resource attribute value using one of the special functions DATE, DAY, or TIME as described below.

ACCEPT Examples

```
ACCEPT STARTING_VALUE
ACCEPT TODAYS_DATE FROM DATE
```

12.5.1.1 DATE Option

This option creates an eight-digit number representing the current date, and places it in the named attribute. The attribute type must be INTEGER. The first four digits represent the year, the next two represent the month and the last two give the day. July 14, 1984, for example, would be 19840714.

12.5.1.2 DAY Option

This option creates a five-digit number representing the current date, and places it in the named attribute. The attribute type must be INTEGER. The first two digits represent the year and the remaining three represent the day number within the year. July 14, 1984, for example, would be 84196.

12.5.1.3 TIME Option

This option creates an eight-digit number representing current real time. (Note that this will, in general, be different from the simulated time value represented by CLOCK_TIME). The type of the named attribute must be INTEGER. The first two digits represent the hour (24-hour clock), the next two represent minutes, the next two represent seconds and the last two represent hundredths of a second. Twenty-two minutes 10.3 seconds after 3:00p.m., for example, would be 15221030.

The accuracy of the time will depend on the time accuracy of the machine used (i.e., some machines may not have accuracy of hundredths of a second).

12.5.2 DISPLAY STATEMENT

This statement causes an attribute value or nonnumeric literal to appear immediately on the terminal screen.

$$\text{DISPLAY } \left\{ \begin{array}{l} \text{attribute_name_1} \\ \text{nonnumeric_literal_1} \end{array} \right\} \left[\dots \left\{ \begin{array}{l} \text{attribute_name_n} \\ \text{nonnumeric_literal_n} \end{array} \right\} \right]$$

As many attribute names or literals as required may be listed, separated by commas. When displayed, they will appear directly after each other on one line. The quotation marks surrounding nonnumeric literals will not be displayed.

DISPLAY examples

```
DISPLAY TOTAL_COUNT
DISPLAY 'ENTER STARTING VALUE'
DISPLAY 'TOTALS WERE ', TOTAL_SENT, ' ', TOTAL_RECEIVED
```

12.5.3 PRINT STATEMENT

This statement causes an attribute value or nonnumeric literal to be output to the file `specification_name.LIS` once the task is complete. The `specification_name` is the name of the task control specification used to run the task (refer to Chapter 10).

$$\text{PRINT [START_PAGE]} \left\{ \begin{array}{l} \text{attribute_name_1} \\ \text{nonnumeric_literal_1} \end{array} \right\} \left[\dots \left\{ \begin{array}{l} \text{attribute_name_n} \\ \text{nonnumeric_literal_n} \end{array} \right\} \right]$$

As many attribute names as required may be listed, separated by commas. When printed, they will appear directly after each other on one line. Each PRINT statement causes a new line to be started on the output file. If the optional START_PAGE keyword is used a new page is started before printing the specified items.

12.5.4 TRACE STATEMENT

This statement operates like a PRINT statement, placing specified resource attribute values on the special trace file 'specification-name.TRC'. Unlike a PRINT statement, however, a TRACE statement may be selectively activated without changing the process in which it appears. This facility is particularly useful when detailed examination of the changing values of attributes may be required, for example when debugging.

$$\text{TRACE [START_PAGE]} \left\{ \begin{array}{l} \text{attribute_name_1} \\ \text{nonnumeric_literal_1} \end{array} \right\} \left[\dots \left\{ \begin{array}{l} \text{attribute_name_n} \\ \text{nonnumeric_literal_n} \end{array} \right\} \right]$$

To activate a TRACE statement, the task control specification must contain the keyword TRACE (see Section 13.1.5). The code for these TRACE statements will only be incorporated with the prepared process when one of the TRACE options is selected when preparing the process. One can turn the trace facilities ON and OFF during the course of a simulation by using the following statements in a process. The TURN TRACE ON statement is *required when tracing processes that are incorporated into an RTG task*, such as a background overlay module, since the RTG task is not started from the control specification.

$$\left[\begin{array}{l} \text{TURN TRACE ON} \\ \text{TURN TRACE OFF} \end{array} \right]$$

These statements facilitate producing short traces at specific times or about specific functions that would otherwise generate large trace outputs. They are required when tracing graphics processes.

12.5.5 READ AND WRITE SEQUENTIAL FILE STATEMENTS

These statements enable sequential data records, structured by resources, to be read and written from/to an external file. They may only be used in a process, which interfaces with that file. Sequential data files may be used as input to a simulation run, or created as output from a simulation run. The maximum record size is 32,000 bytes for any VSE sequential file. To access a VSE file, the user must create a resource that corresponds to the record structure of the file.

File icons are used to connect an external resource to the desired file, whether it is used for input or output. Examples of this are shown in Figure 12-4. The external input file, INDEX_INPUT, will be connected to the resource, INDEX_INPUT_INTERFACE, and the external output file, INDEX_OUTPUT, will be connected to the resource, INDEX_OUTPUT_INTERFACE.

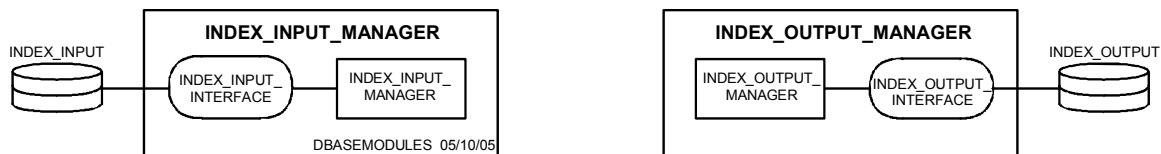


Figure 12-4. Examples of external input and output files.

Data may be read using the keywords OPEN and READ, and written using the keywords OPEN and WRITE. These files may contain the following types of records:

FIXED Length Sequential

The FIXED length record format provides for sequential access to fixed length records created by previous simulation runs on the same platform. It can also be used to transfer data across platforms, provided that the data is in readable or printable character form, or the platforms are compatible. If these files contain integers, real, double precision real, etc. type attribute data, where the internal binary representation of these attributes typically falls outside the range allowed by the system editor or printer, then they would produce unpredictable results if edited, printed, or read on another platform.

VARIABLE Length Sequential

The VARIABLE length record format provides for sequential access to variable length records created by previous simulations on the same platform. Variable length records can be used to compress large databases. This format can also be used to transfer data across platforms, provided that the data is in readable or printable character form, or the platforms are binary compatible. If these files contain integers, real, double precision real, etc. type attribute data, where the internal binary representation of these attributes falls outside the range allowed by the system editor or printer, then they would produce unpredictable results if edited, printed, or read on another incompatible platform, unless the user knows the translation codes.

Variable length files are of two types:

- USER KNOWN - It is up to the user to read the correct number of bytes.
- AUTOMATIC - The system records and reads the number of bytes automatically.

TEXT Sequential

The TEXT format provides for sequential access to records on files created by the keyboard editors for that platform and operating system, or created by previous simulation runs using TEXT file format. TEXT files must contain only printer recognizable characters or text type data, e.g. attributes of type character or decimal (as opposed to binary integer or real data). These files may contain records of variable length, separated by an end-of-record designator for the platform, typically a line feed character.

12.5.5.1 WRITE Sequential File Statement

To WRITE to a Sequential File, one writes the resource connected to that file (only one can be connected). This is done using the following WRITE statement.

```
WRITE resource_name [[WITH] SIZE = size_attribute] [WITHOUT LINEFEED]
```

The named resource must have been specified as EXTERNAL in the resource list for the process containing this statement (refer to Section 3.8.2). For sequential files (FIXED, VARIABLE, or TEXT), the WRITE statement copies the named resource into the next sequential record for the external file. For FIXED length record files, the record size matches the resource size, and no delimiters exist between records in the file. Neither the SIZE = option or the WITHOUT LINEFEED option apply. Fixed length record files must be read using the same fixed length record size used when they were written, reference Section 12.5.5.2.

For VARIABLE length record files, the user must specify whether the external resource record length will be AUTOMATIC (A) or USER KNOWN (U) when read (refer to Section 4.1.3). If the SIZE = size_attribute option is not used, the default SIZE will be the size of the resource, which is also the maximum record size. Otherwise the record SIZE is determined by the size_attribute in the WRITE statement. The maximum value of SIZE is limited to 32,000, the maximum record size in bytes. When the AUTOMATIC option is used, the record SIZE is recorded in a special two byte field ahead of the physical record; this SIZE is then available upon reading a variable length record file, reference Section 12.5.5.2. The WITHOUT LINEFEED option does not apply to variable length records.

For TEXT files, both the SIZE = size_attribute option, and the WITHOUT LINEFEED option may be used. If the WITHOUT LINEFEED option is *not* used, the record will be one byte greater than the specified SIZE or resource size, and will contain the "line feed" character recognized by the system editor and printer of the platform. If the WITHOUT LINEFEED option is used, then no line feed character will be inserted after the record, and no delimiter will exist between it and the subsequent record.

If the `SIZE = size_attribute` option is used, the record will be the length specified by the `size_attribute`. If it is not used, then the record size defaults to the resource size.

WRITE Sequential examples

```
WRITE OUTPUT_MESSAGE
WRITE TRANSMISSION_RECORD WITH SIZE = MESSAGE_LENGTH
WRITE PRINTER_RECORD WITH SIZE = ESCAPE_SEQUENCE_SIZE
WITHOUT LINEFEED
```

12.5.5.2 READ Sequential File Statement

To READ to a Sequential File, one reads the resource connected to that file (only one can be connected). This is done using the following READ statement.

```
READ resource_name [ [WITH] size_attribute = SIZE
                    [WITH] SIZE = size_attribute ]
AT_END statement
```

The named resource must have been specified as EXTERNAL in the resource list for this process (refer to Section 3.8.2). For sequential files (FIXED, VARIABLE - AUTOMATIC, VARIABLE - USER KNOWN, or TEXT), the READ statement copies the next sequential record from the external file into the named resource. For FIXED length record files, the record size must match the resource size, and neither SIZE option applies.

For VARIABLE length record files, the user must specify whether the incoming record lengths will be AUTOMATIC or USER KNOWN. If USER KNOWN, the `SIZE = size_attribute` must be used, else the default size will be the size of the resource, which cannot exceed the maximum record size of 32,000 bytes. If AUTOMATIC is used, the record SIZE is automatically read from a two-byte field ahead of the physical record. The user may use the `size_attribute = SIZE` option to obtain the size of each record read. If the record size is smaller than the external resource size, spaces will be inserted; if the record size is larger, it will be truncated.

For TEXT files, records are delimited by the recognized linefeed character of the file, and can be of variable length. Use of the `size_attribute = SIZE` is optional, and is provided so the user can obtain the size of each record read. If the record size is smaller than the external resource size, spaces will be inserted; if the record size is larger, it will be truncated.

The AT_END clause must be used to control what happens when the process attempts to read past the end of the external file. Only the first simple statement following the AT_END keyword will be in the AT_END clause, and this cannot be a conditional statement.

READ Sequential examples

```
READ EXTERNAL_FILE
AT_END EXECUTE SYNTAX_CHECK

READ EDITOR_FILE WITH RECORD_LENGTH = SIZE
AT_END SET FILE_STATE TO END_OF_FILE
```

12.5.6 READ AND WRITE DIRECT ACCESS FILE STATEMENTS

These statements enable direct access data records, structured by resources, to be read and written from and to an external file. They may only be used in a process, which interfaces with that file.

Direct access files are accessed as INPUT_OUTPUT, and records may be read and written in any desired order using a relative access key (refer to Sections 12.5.6.1 and 12.5.6.2). These files can only be accessed by GSS on the same platform and operating system on which they were originally created.

Before a direct access file can be written or read as INPUT_OUTPUT, it must first be created so that the relative record keys used by the record_id are known to the operating system. The procedure for creating a direct access file is described in Section 12.5.6.3.

12.5.6.1 WRITE Direct File Statement

To WRITE to a Direct Access File, one writes the resource connected to that file (only one can be connected). This is done using the following WRITE statement.

```
WRITE resource_name RECORD [TO] record_id
      [IF] INVALID_RECORD statement
```

For Direct Access files, the WRITE statement copies the named resource into the record specified by record_id; the fixed-length record size will match the resource size. Record_id must be either type INTEGER, INDEX or INDEX_1 and defines the record number relative to the beginning of the file. The INVALID_RECORD clause must be used to control what happens when the process attempts to WRITE to a record_id (key) that does not correspond to a record on the external file, reference Section 12.5.6.3. Only the first statement following the INVALID_RECORD keyword will be in the INVALID_RECORD clause.

WRITE Direct Access File example

```
WRITE NEEDLINE_DATA RECORD TO NEEDLINE_NUMBER
      IF INVALID_RECORD EXECUTE ERROR_PROCEDURE
```

12.5.6.2 READ Direct File Statement

To READ to a Direct Access File, one reads the resource connected to that file (only one can be connected). This is done using the following READ statement.

```
READ resource_name RECORD [FROM] record_id
    [IF] INVALID_RECORD statement
```

For direct files, the READ statement copies the record specified by `record_id` into the named resource; the fixed-length record size must match the resource size. `Record_id` must be either type INTEGER, INDEX or INDEX_1 and defines the record number relative to the beginning of the file. The INVALID_RECORD clause must be used to control what happens when the process attempts to read using a `record_id` (key) that does not correspond to an existing record on the external file, reference Section 12.5.6.3. Only the first statement following the INVALID_RECORD keyword will be in the INVALID_RECORD clause.

READ Direct example

```
READ NEEDLINE DATA RECORD FROM NEEDLINE_NUMBER
    IF INVALID_RECORD EXECUTE ERROR_PROCEDURE
```

12.5.6.3 CREATING a Direct Access File

Before a Direct Access File can be accessed, it must be created. To do this, one must build a special *create* utility tailored to create a new file. This utility must put the desired number of records onto the file. If the file size is to be changed, then a new file must be created, and the old file contents copied into the new file. With this in mind, one should allow for the maximum number of records anticipated when creating the file.

When creating a direct access file, it must be opened as OUTPUT, *not* INPUT_OUTPUT. The external resource describing the record on the file must have its record type declared as DIRECT. To create the file, one simply writes records sequentially to the file using the WRITE statement below.

```
WRITE resource_name RECORD
```

The size of the direct access file is determined by the number of records written. If a direct access file is required that can store a maximum of 100,000 records, then the user must create this file by writing 100,000 records sequentially during the create process.

12.5.7 OPEN STATEMENT

Before an external file may be used by a process, it must be named in an OPEN statement. This serves to open the appropriate external file.

$$\text{OPEN} \left\{ \begin{array}{l} \text{INPUT} \\ \text{OUTPUT} \\ \text{INPUT_OUTPUT} \end{array} \right\} \text{resource_name}$$

For sequential files, the keyword OPEN is used with either the word INPUT or OUTPUT to indicate the usage of a resource within the process. INPUT resources may subsequently be read, and OUTPUT resources may subsequently be written. A sequential file may not be both read and written on the basis of one OPEN statement.

For direct access files, the keyword OPEN is used with the word INPUT_OUTPUT to indicate that relative type external resources may be either read or written directly, except when they are being created, in which case they must be opened as OUTPUT.

OPEN examples

```
OPEN INPUT INPUT_MESSAGE
OPEN OUTPUT TRANSMISSION
OPEN INPUT_OUTPUT NEEDLINE_FILE
```

12.5.8 CLOSE STATEMENT

This statement terminates the effect of the most recent OPEN statement for a specific named resource. Its effect is to close the external file.

```
CLOSE resource_name
```

No READ or WRITE statements may be used for this resource once the CLOSE statement is written. However, the same resource may be accessed more than once in the same process, possibly with different purposes (INPUT or OUTPUT). Each OPEN statement, however, must be paired with a CLOSE statement.

CLOSE example

```
CLOSE NEEDLINE_FILE
```

12.5.9 DYNAMIC FILE ASSIGNMENT

When running tasks that read and write various device resources, e.g., data files on disk or tape, communication channels, the keyboard and screen, it is necessary to describe the external device or file record formats as GSS EXTERNAL resources. The names of these external resources are then used internally in READ and WRITE statements.

In an on-line interactive environment, it is desirable to make these file assignments dynamically, so that new files can be created and read while the task is running. One may also want to access different files depending upon input to the simulation at the time. To perform these functions, it is necessary to be able to make dynamic external resource assignments within a process, and to test the existence and contents of these resources. The dynamic file assignment statement links the external resources and the external files during execution of the process by equating their names. The format for this statement is as follows:

$$\text{ASSIGN } \left\{ \begin{array}{c} \text{file_name} \\ \text{attribute_name} \end{array} \right\} \text{ TO external_resource_name}$$

In this statement, file_name must contain a nonnumeric literal, of up to 40 characters, that represents a fully qualified file_name, including a path_name/file_name within the context of the users directory and the operating system conventions. If no path_name is specified, the users current directory is the default. The file_name itself must satisfy the definition described under EXTERNAL RESOURCE.

Example of assigning files to external devices

```
IF MY_EXTERNAL_FILE EXISTS
    AND MY_EXTERNAL_FILE IS NOT EMPTY
        ASSIGN MY_EXTERNAL_FILE TO EXTERNAL_RESOURCE_NAME.
.
.
.
OPEN INPUT EXTERNAL_RESOURCE_NAME
READ EXTERNAL_RESOURCE_NAME
```

12.6 COMMUNICATIONS CHANNEL STATEMENTS

The External TCP/IP Communications Channel Resource and Communications Channels are described in the GSS and VSE User's Manuals. This section describes the process statements associated with these Communications Channels. Use of these process statements makes communications over a TCP/IP connection easy and straight-forward and avoids the complex, low-level details usually associated with using TCP/IP that are encountered in other languages.

12.6.1 SEND STATEMENT

The SEND statement is used to send information over a TCP/IP Channel after it has been opened. When the WITH SIZE clause is omitted, the resource size is used. The ON_FAIL condition would be true if, for example, a client attempted to send, and the server had gone down. In this case the GSS statement following ON_FAIL would be executed.

```
SEND resource_name
    [WITH SIZE {numeric_literal }
      {numeric_attribute}]
    [ON_FAIL statement ]
```

12.6.2 RECEIVE STATEMENT

The RECEIVE statement is used to receive data coming over the TCP/IP Communications Channel. When the ON_INSUFFICIENT_DATA option is used, the receive is non-blocking; if used in conjunction with the WITH SIZE clause, attribute_name will return the number of characters received. When the ON_INSUFFICIENT_DATA option is omitted, the receive is blocking and hangs until the requested amount of data is available.

```
RECEIVE resource_name
    [WITH SIZE {numeric_literal }
      {numeric_attribute}]
    [ON_INSUFFICIENT_DATA statement ]
    [ON_FAIL statement ]
```

The ON_FAIL option for RECEIVE is not implemented in the current release.

12.6.3 IF EXISTS STATEMENT (Not implemented in the current release)

Since TCP Communication Channels require connections, they have to be opened before they can be used. The IF EXISTS syntax tests whether or not a TCP connection has been established and is currently open for a given channel.

```
IF resource_name CHANNEL { EXISTS
                           DOES NOT EXIST }
```

12.6.4 CONNECT STATEMENT

Since TCP/IP Communication Channels require connections, they cannot be used before they are established. The connect statement is used to setup these connections. One form is used for Clients and another form is used for Servers. On either form, when the TIME_OUT option is used, TCP opens will return after TIME_OUT seconds if unsuccessful and will set the ON_FAIL condition to true. Otherwise, unsuccessful TCP opens will hang indefinitely.

If the server_name or ip address is not provided, the CLIENT channel will be connected to the server specified when the channel is created. If the port_number is not provided, the channel will be connected to the port specified when the channel is created.

```
CONNECT CLIENT resource_name
    [TO SERVER {server_name
               ip_address} [TO] [PORT ip_port_number]]
    [TIME_OUT {numeric_literal
              numeric_attribute}]
    [ON_FAIL statement ]

where server_name = {non_numeric_literal
                    non_numeric_attribute}
      ip_address  = {non_numeric_literal
                    non_numeric_attribute}
      ip_port_number = {numeric_literal
                       numeric_attribute}
```

For the Client, server_name can take the form 'MY_SERVER.COM'. Use of this form requires that the platform running GSS have access to a Domain Name Server (DNS) to translate the supplied server_name to an IP address.

The server_name can be supplied as either a non_numeric_literal as in the example given, or by virtue of a reference to a non_numeric_attribute, e.g., SERVER_NAME. The Client CONNECT statement can also accept a conventional IP address, e.g., 135.23.167.52, which can be supplied as a non_numeric_literal or as a non_numeric attribute, e.g., IP_ADDRESS_ATTRIBUTE.

Also both Client and Server can take an optional IP Port number in either numeric literal form or by reference to a numeric attribute. To connect the server to a port, the following format is available.

```
CONNECT SERVER resource_name
    [TO PORT ip_port_number]
    [TIME_OUT {numeric_literal
               {numeric_attribute}}]
    [ON_FAIL statement]

where ip_port_number= {numeric_literal
                      {numeric_attribute}}
```

12.6.5 CLOSE STATEMENT

When a TCP/IP channel is closed, the connection it represents is closed. It can be reopened with the OPEN syntax.

```
CLOSE resource_name
```


12.7 SYSTEM LEVEL STATEMENTS

System Level statements are calls to the operating system (OS). To make use of these statements, they must be recognized by the OS. VPOS is designed to recognize these statements.

12.7.1 EVENT SYNCHRONIZATION (WAIT UNTIL) STATEMENT

As defined in Chapter 11 Section 11.4.3.6, EVENTS occur at the system level and are handled by the OS. Events in one task or IND module can be used to change the state of another task or IND module such that the actions of both are synchronized. Synchronization is accomplished by the WAIT UNTIL statement whose format is defined below.

```
WAIT UNTIL event_name IS alias_name
```

The WAIT UNTIL synchronization statement causes a process in an IND Module or task to WAIT UNTIL a particular EVENT state has taken on a specified value. To operate properly, the process containing the WAIT statement must be restarted by the OS in a manner that is synchronized with processes on different processors in the same task, or processes in other tasks. Thus the OS implementation must be designed to support an intricate and immediate succession of such statements.

The following is an example of the WAIT UNTIL statement.

Example

```
IF INPUT_EVENT IS STOP  
    WAIT UNTIL INPUT_EVENT IS GO .
```

The above statement implies that, when the INPUT_EVENT state is changed to GO by another process, the process in the WAIT state will continue immediately thereafter on a synchronized basis determined by the application system design.

12.7.2 FREE MEMORY RESOURCE STATEMENT

The FREE resource_name MEMORY statement tells the OS that the named resource will no longer be used by the task. This statement does not apply to IT or IP resources.

```
FREE resource_name [MEMORY]
```

When this statement is executed, the OS frees the memory previously assigned to this resource, and it becomes immediately available for reassignment. The following is an example of the FREE MEMORY statement.

Example

```
FREE SHARED_RESOURCE_5 MEMORY
```

12.7.3 RELEASE OF AND ACCESS TO IP RESOURCES

These statements only apply to processes that write or read IP resources. An IP resource must reside within an IND module, and may only be written by processes within that same IND module. It may be read by multiple processes, including those outside the IND module in which it resides.

When a process writes to an IP resource, that resource is *automatically* released to those that read it immediately after the process that writes it has completed. Processes that read an IP resource are *automatically* given access to it immediately before they start to run.

In some cases, a process that writes to an IP resource may run for a while after it puts data into the IP resource, e.g., when reading or writing files,. In these cases, it may want to RELEASE the IP resource prior to its completion so that processes reading it from outside that IND module have access to the latest copy. If it is updating the IP resource multiple times while it runs, taking time to perform other processing in between, it may want to RELEASE the resource multiple times while it is running, so that other processes can access the latest copies that are produced.

Conversely, processes that are reading IP resources that reside outside their IND module may be taking time to perform processing before accessing them, and therefore may need to ACCESS the latest copies while performing substantial processing in between. In this manner, a process can ensure having ACCESS to the latest copy of an IP resource immediately before its use. To support these functions, the following RELEASE and ACCESS statements are available to the corresponding writers and readers.

```
RELEASE IP_resource_name  
ACCESS  IP_resource_name
```

The above synchronization actions are often used with the Event Synchronization statement (WAIT UNTIL). The use of these statements is easily determined by application experts who understand the underlying need for synchronization at the application level.

13. TASK CONTROL SPECIFICATION

The content of the Control Specification is outlined in Figure 13-1 below.

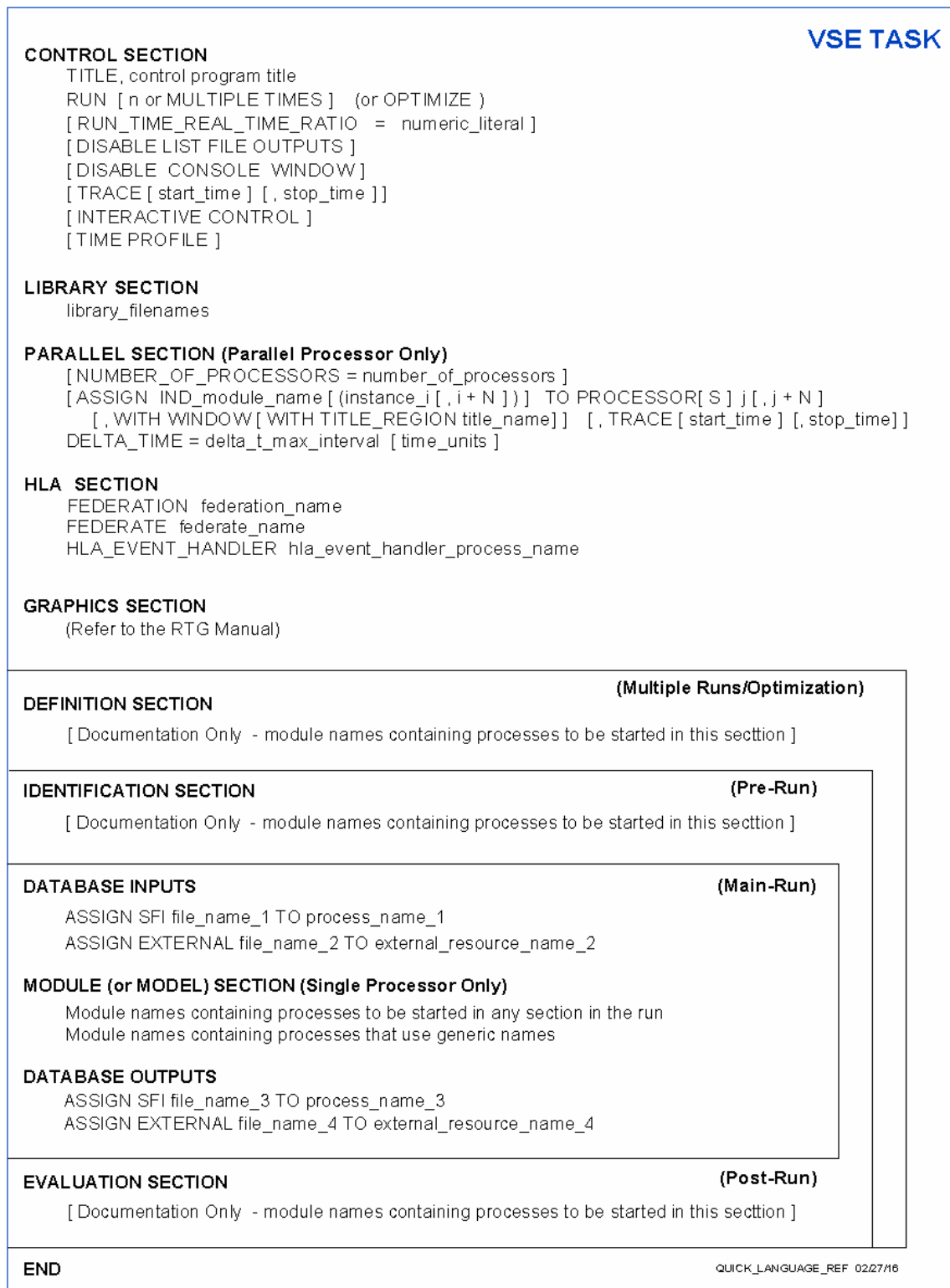


Figure 13-1. Skeleton structure for a control specification.

This skeleton structure shown in Figure 13-1 is provided automatically under the Architecture Subsystem. Each section is described below. Precede comments with ***.

13.1 CONTROL SECTION

This section is used to specify top level controls for the run. These controls are embodied in the following statements.

13.1.1 TITLE, control_specification_name

This statement provides a long title for the run which will appear on all output. The format is as shown above. The control specification title may consist of any combination of VISISOFT characters, up to a maximum of 48. Double quotes may not be used.

13.1.2 RUN_TYPE

This statement determines whether multiple runs are to be invoked. The keyword RUN must be used. The statement format is:

Format	
RUN	<div style="border-left: 1px solid black; border-right: 1px solid black; padding: 0 10px; display: inline-block;"> n TIMES MULTIPLE_TIMES OPTIMIZE </div>

where n is a positive integer. If the keyword RUN is used by itself, VisiSoft assumes that only one run is required. For n > 1, VisiSoft will execute as many runs as specified, reinitializing the run time clock (CLOCK_TIME) before beginning each run.

The multiple run option is set up so each run generates a different statistic. This is done by providing a different random number seed at the beginning of each run. To reset the seed so that each run starts with the same seed, insert the following statements when initializing each run:

```
SEED                CHAR 12
MOVE SPACES TO SEED
ANY_ATTRIBUTE = RANDOM(SEED)
```

where ANY_ATTRIBUTE is any numeric attribute (e.g., DREAL) available to the process containing the statement. RANDOM is the random number generator (defined as DREAL in VisiSoft), and SEED stores the random number generator seed for the next call. One may reset the random number seed at any time during a run by moving spaces to SEED. One may also store multiple SEEDs to maintain different sequences of random numbers. See Appendix 2 for more information.

If the keywords RUN MULTIPLE TIMES are used then, at run time, VisiSoft will prompt for the number of runs before beginning the run. This feature allows the number of runs to be easily varied without repreparing the control specification each time.

Both the current run number and the number of runs to be run may be accessed in a process by using the reserved words RUN_NUMBER and NUMBER_OF_RUNS. Process statements should not assign values to either of these reserved words. These can be used in a conditional statement as follows.

```
IF RUN_NUMBER IS EQUAL TO NUMBER_OF_RUNS  
EXECUTE COMPUTE_FINAL_TOTALS.
```

13.1.3 OPTIMIZE

This statement is used in lieu of RUN when the optimization option is selected. The user is referred to the VisiSoft Optimization User's Manual. This facility contains an easy-to-use constrained nonlinear optimization system with excellent convergence properties. It can be used to identify the optimal values for design parameters as well as the optimal parameters for fitting models to test data. It also allows users to define tolerance ranges on any of the parameters so that constraints can be evaluated based on realistic worst case tolerance variations around the nominal or optimized values.

13.1.4 SIM_REAL_RATIO = numeric_literal

The SIM_REAL_RATIO is used when a run runs faster than real-time. This statement constrains the speed of the run when the SIM_REAL_RATIO provided by the user becomes larger than the actual ratio. Based upon internal system comparisons over small time intervals, the run waits for the real-time clock to catch up. The numeric_literal must be a positive number.

13.1.5 TRACE [START_TIME = trace_start_time] [STOP_TIME = trace_stop_time]

If it is desired to use the trace facility, the keyword TRACE must appear in a separate statement in the CONTROL SECTION. When the control specification is selected for a run, VisiSoft will display the TRACE option box. At that point it is possible to decline the TRACE option for a particular run without changing the control specification. If the user selects the START_TIME, or STOP_TIME option, then these times will be invoked. If either one of START_TIME or STOP_TIME is left out, the beginning or end of the run is substituted respectively. If neither is present, the system will prompt for start time and stop time just prior to the run. If the TRACE option is activated, all TRACE output will be stored in the "file specification-name.TRC" for access via the system editor.

13.1.6 DISABLE LIST FILE OUTPUTS

The DISABLE LIST FILE OUTPUTS option is used to disable the list file. When this option is used, the file will not be opened. If not disabled, the file specification-name.LIS will contain Run-time warning/error messages, boundary check messages and outputs generated by PRINT statements.

13.1.7 DISABLE CONSOLE WINDOW

The DISABLE CONSOLE WINDOW is used to disable the console window. When this option is used, the window will not be opened.

13.1.8 TIME PROFILE

If it is desired to use the time profile facility, the keyword TIME PROFILE must appear as a separate statement in the CONTROL SECTION. When the task control specification is selected for execution, the user will be prompted to determine if the profile option is to be activated. At that point it is possible to decline the profile option for a particular run without changing the control specification. If the profile option is activated, all profile output will be stored in the file "specification-name.PRF" for access via the system editor or query functions.

13.2 LIBRARY SECTION

This section allows the user to specify the names of libraries to be used when linking a task that uses library modules. Simply list the names of the files containing the libraries to be used, one per line. The path can also be specified as "*path / file_name*". The number of characters is limited to 256.

13.3 PARALLEL SECTION

This section is used to specify top level controls for the run. These controls are embodied in the following statements.

13.3.1 NUMBER OF PROCESSORS

This statement allows the user to specify the number of processors to be used for the run. The format is as shown below. The number of processors may be specified as an integer number.

Format	
NUMBER_OF_PROCESSORS =	$\left\{ \begin{array}{c} n \\ \text{number_of_processors} \end{array} \right\}$

13.3.2 IND MODULE ASSIGNMENTS

This statement allows the user to specify the IND Modules to be used in the run. The format is as shown below. Using this statement, a named ind_module may be placed on a specified processor number. This statement is repeated to define the number of IND Modules to be used in the run.

Format
ASSIGN ind_module_name [(instance_i [, i+N])] TO PROCESSOR[S] j[, j+N] [, WITH WINDOW [, WITH TITLE_REGION title_name]] [, TRACE[start_time] [, stop_time]]

The top option provides for group assignments of IND module instances to processors. The following lines illustrate the assignment option with examples illustrated in Figure 13-2.

- The first option is a single IND module to processor assignment. As illustrated in the first examples, this gets cumbersome as the number of module instances becomes large.
- The second option assigns instances i thru i + N to processors j thru j + N.
- The third option assigns instances i thru i + N to processor K;
instances j thru j + M to processor L; and
instances k thru k + P to processors q thru q + P.

The box below contains an example of code:

```
PARALLEL SECTION
NUMBER_OF_PROCESSORS = 10
ASSIGN IND_SCAN_PARTITION_X      TO PROCESSOR 1
ASSIGN IND_SCAN_PARTITION_Y      TO PROCESSOR 2
ASSIGN IND_SCAN_PARTITION_Z(1)   TO PROCESSOR 3
ASSIGN IND_SCAN_PARTITION_Z(2, 8) TO PROCESSOR 4, 10
```

Figure 13-2. Examples of IND module to processor assignments.

Also, WINDOWS may be opened separately for each processor, each with its own title.

Finally, TRACES may be specified with start times and end times for each IND Module independently as described in Section 13.1.5 above.

13.3.3 DELTA_TIME ASSIGNMENT

This statement allows the user to specify the Delta_Tmax_interval to be used by the Run-Time System synchronizers to ensure synchronization with the specified interval for the run. The format is as shown below.

Format
DELTA_TIME = delta_t_max_interval

13.4 HLA SECTION

This section is used to specify the top level controls for a run which is participating as a federate in an HLA federation. The format for these control statements is as follows:

```
HLA SECTION
  FEDERATION federation_name
  [ FEDERATE federate_name ]
  [ HLA_EVENT_HANDLER process_name ]
```

The “federation_name” specifies the federation to which the current VisiSoft run/federate will join. After the run/federate joins the federation, the run/federate will be identified by the “federate_name” in the federation name space. If “federate_name” is not specified in this section, run name will be used as federate name.

The HLA_EVENT_HANDLER clause identifies the process_name to be invoked automatically when an HLA PUBLISH event for one of the HLA resources in the VisiSoft run has occurred. The process defined as process_name will automatically be invoked by the VisiSoft system only when an HLA resource is updated by the RTI/federation. In this case, the HLA event handler process will be called before the next scheduled or called process is executed. The updated HLA resource name is stored in the HLA_UPDATE_RESOURCE system attribute. This attribute is useful when there are multiple HLA resources defined in a run because it will identify the specific resource updated for the current PUBLISH event.

13.5 GRAPHICS SECTION

This section is used to invoke the VisiSoft Run-Time Graphics facilities. Refer to the RTG Users Manual for an explanation of this section.`

13.6 SECTIONS FOR STARTING PROCESSES

When starting a run, independent modules are started when one or more of the processes in that module are started. As illustrated in Figure 10.1, the DEFINITION, IDENTIFICATION, MODULE, and EVALUATION sections provide for starting processes at multiple points in a run. *The processes to be started are identified in the architectural drawing.* These sections in the CONTROL SPECIFICATION are used for documentation only. These sections are invoked by the architecture to start processes and corresponding modules as follows:

DEFINITION SECTION - In the case of multiple runs, processes started in this section are executed only once, immediately prior to the first run. This section is used to start overall initialization and set constants, items that need not be repeated each time another run begins. The run-time clock, `CLOCK_TIME`, is initialized to zero at the start of the Definition Section.

IDENTIFICATION SECTION - In the case of multiple runs, processes started in this section are executed at the start of each run. This is the place to start initialization processes that zero counters and summary attributes to be output for each run. Processes in this section can be used to set values depending on the results of a previous run. This supports parametric or sensitivity analysis, and optimization. The value of the clock at the beginning of this section will equal its value at the end of the Definition Section (and will be reset to this value in the case of multiple runs).

MODULE SECTION - Processes started in this section represent the main body of the run. The value of the run-time clock at the beginning of this section will equal its value at the end of the previous section.

EVALUATION SECTION - In the case of multiple runs, processes started in this section are used to evaluate results of each run once it is completed. Typically these processes involve computation of performance measures or summary results such as averages from the run. When using the VisiSoft Optimization option, this section can be used to start processes that evaluate constraints and optimization criteria, functions that can only be evaluated when a run has completed. The value of the run-time clock at the beginning of this section will equal its value at the end of the MODULE section.

Processes started in any section may **SCHEDULE** or **CALL** other processes, and execution will continue, advancing the run-time clock if necessary, until no more processes are left to be executed, or until a **STOP** statement is reached (see Section 9.4.5). The run will then advance to the next section which has designated started processes, or terminate.

As explained in the Definition, Identification, Module, and Evaluation sections below, the Module Section must be used to list the module names that contain processes to be started. Processes can be started in any section by designating these sections at the time of process creation or modification. The contents of the Definition, Identification, and Evaluation sections are for documentation purposes only and are free format. In the case of a module containing two or more processes that are started in *different* sections, the user may find it convenient to provide appropriate notes, listing processes that are started, in the appropriate sections.

13.7 DEFINITION SECTION

As explained in Section 13.6, this section is for documentation purposes only.

13.8 IDENTIFICATION SECTION

As explained in Section 13.6, this section is for documentation purposes only.

13.9 DATABASE INPUTS

This section may be used to reassign external files to an external resource, or to invoke the Standard File Interface (SFI) option for input data to a task. SFI is a set of standards for interchanging data files that typically have a large number of samples, wherein each sample has a number of data elements (up to 24). Samples may be ordered by time or frequency, or sample number, or some other user-selected identification (ID field). To meet SFI interchange requirements, these files must have standard header and format information on two successive records with prescribed record formats. The data must also follow the SFI record format described in Chapter 14.

The format for SFI database assignment statements is as follows:

```
ASSIGN SFI path/file_name TO process_name
```

Here process_name refers to a process written to accept input from an SFI file named file_name. The database assignment statement should start in or beyond column 7.

Whenever an SFI input process is called or scheduled, the attributes named in this section will be updated automatically from the corresponding series fields on the SFI file.

Up to 40 SFI files may be referenced as input in a task control specification, with the limit that the total input and output files cannot exceed 80. The SFI files must be created and data entered before beginning a VSE task.

NOT IN THIS RELEASE

The statement for reassigning external files to external resources in the database inputs section is as follows:

```
ASSIGN path/file_name TO external_resource_name
```

Here external-resource-name refers to an external resource already attached to a process written to accept input from the named file.

Examples

```
DATABASE INPUTS
```

```
ASSIGN SFI /usr/psi/NEEDLINE_FILE TO NEEDLINE_INPUT
ASSIGN SFI ACTIVITY_DATA_FILE TO ACTIVITY_INPUT
ASSIGN ACCOUNT_FILE TO EXTERNAL_ACCOUNT_RECORD
```

13.10 MODULE SECTION

This section applies when the Parallel Processor Section is not used. The contents of the Module Section must be module names, one name per line. Module name entries can be hierarchical or elementary, and will be resolved to the processes in elementary modules with their *start flags* set. These processes will be started in those run sections that were designated at the time they were created.

The order in which processes are started within a run section is based on the order of their respective module names in the Module Section. Since the submodules and processes of a module hierarchy have no particular order, the order of started processes *within* a module entry is undefined. If this must be resolved to ensure run validity, one can start a single process that starts the other processes, or use the priority code associated with each started process.

In addition to module entries containing started processes, the Module Section can also be used to resolve Generic Names. Modules specified for purposes of resolving Generic Names may optionally contain started processes. Resolution of references to process generic names can be accomplished by listing the modules containing those processes in this section.

A Module Section entry that - (1) has no matching module name in the user's directory; or (2) matches that of a module with no process starting flags set and no generic name - will cause a severe error message to be issued, and the run will not be started.

In summary, module names are listed in the Module section to have their started processes included in a designated control section's start list, and to resolve generic process names.

13.11 DATABASE OUTPUTS

This section may be used to reassign external files to an external resource, or invoke the SFI option for output data from a task. As explained in the prior section on database inputs, SFI is a set of standards for interchanging data files that typically have a large number of samples. To meet SFI interchange requirements, these files must have standard header and format information on two successive records with prescribed record formats. The data must also follow the SFI record format.

The format for SFI database assignment statements is as follows:

```
ASSIGN SFI path/file_name TO process_name
```

Here, process_name refers to a process written to produce output for an SFI file named file_name. A process may provide output to exactly one SFI file, and an SFI file may only be linked to one process during a run. The database assignment statement should start in or beyond the 7th column.

Whenever an SFI output process is called, the values of the named attributes will be placed automatically in the corresponding data fields on the SFI file just before exit from the named process. Up to 24 SFI files may be referenced as output in the task control specification.

Not In This Release

The statement for reassigning external files to external resources in the database outputs section is as follows:

```
ASSIGN file_name TO external_resource_name
```

Here external_resource_name refers to an external resource to be attached to the named file "file_name" and written as output from the process.

Examples

```
DATABASE OUTPUTS
```

```
ASSIGN SFI NEEDLINE_FILE TO NEEDLINE_INPUT  
ASSIGN SFI ACTIVITY_DATA_FILE TO ACTIVITY_INPUT  
ASSIGN ACCOUNT_FILE TO EXTERNAL_ACCOUNT_RECORD
```

13.12 EVALUATION SECTION

As explained in Section 13.6, this section is for documentation purposes only.

13.13 END

This must always be the last line in a Control Specification.

The file specification_name.LIS will contain a list of the input and/or output SFI files/series and the associated processes/attributes. In addition, the files specification_name.LIS will also contain a list of external resources and the associated external files.

If there are any diagnostic messages during the Task run because of conflicts etc., a message will be displayed on the screen for each occurrence and will also be echoed on the file specification_name.DIA for access via the system editor.

If any run-time errors occur during the task, they will be displayed on the screen and will also be echoed to the file, specification_name.DIA.

CHAPTER 14

GENERATING A TAILORED RUN-TIME SYSTEM

THE IMPORTANCE OF MATCHING CAD TO PARALLEL PROCESSING

From prior chapters, it should be apparent that there are many difficult concepts to understand and problems to be solved to create a CAD system for parallel processors. These difficult concepts and design approaches are aimed at making it easy for the end-user to create parallel processing architectures as well as provide significant improvements in speed. One must separate these significant differences, i.e., the level of difficulty in development of the CAD system itself, versus the ease of use of the CAD system by subject area experts.

Having evolved a proven theoretical framework derived from many problem types over many years, we have been able to derive important concepts that greatly simplify the solutions. These next three chapters describe approaches required to complete the integration of an effective Software-Hardware Run-Time environment. One is the need for a fully integrated approach as described in Chapter 8. A major contributor to this approach is derived from a little known principle used in certain CAD systems.

The Essence Of The Most Advanced CAD Systems

CAD systems originated in the early 1960s, developed first by computer circuit designers and followed by logic designers. The earliest systems required textual definitions of the problem, including the ability to input large systems of equations. Electronic circuit designers entered element-node connection lists which were used to automatically derive the circuit equations and corresponding matrix solutions.

As the need for circuit reliability and corresponding variational analysis grew, designers were faced with running large numbers of simulations. As circuit sizes became large, speed of a single simulation became important. This led to optimal sparse matrix techniques, of which the fastest by far was the symbolic solution. Using this approach, standard matrix inversion techniques went by the wayside. Instead, the minimum operation count solution was achieved using the “paper and pencil” method for solving matrix equations. Looping was eliminated. The only operations performed were those absolutely needed to get the solution. This virtually eliminated independence in the equations. Almost all equations depended upon those above. Inherent parallelism was virtually eliminated with this extremely fast approach.

Very few CAD packages used this relatively unknown solution. Those that did, kept their approach under wraps and achieved speeds unmatched by competitors. The secret to this approach is to maximize the percentage of the problem that can be solved in the development environment, automatically generating the code representing a very fast, symbolic solution. This code was then compiled and linked with the rest of the run-time software. The time to perform the automatic code generation step was eclipsed by the length of a simulation run.

The Run-Time System (RTS) in VisiSoft follows this same principle. Special code is generated in the development environment that takes full advantage of the critical architectural information necessary to produce highly effective run-time code that provides this information and facilities to VPOS. This code is substantial and complex, with copies on each processor. It is another example of using memory to gain speed. However, the amount of memory used is small compared to what is available today, and likely insignificant relative to what will be available tomorrow. This is another case of trading memory for speed.

The VisiSoft CAD Approach

As illustrated in Figure 14-1, the concepts described above have been incorporated into the VisiSoft CAD system for building and running software applications on parallel processors. This system recognizes the need to automatically generate software tailored for the application, matching a particular software architecture to a physical hardware system.

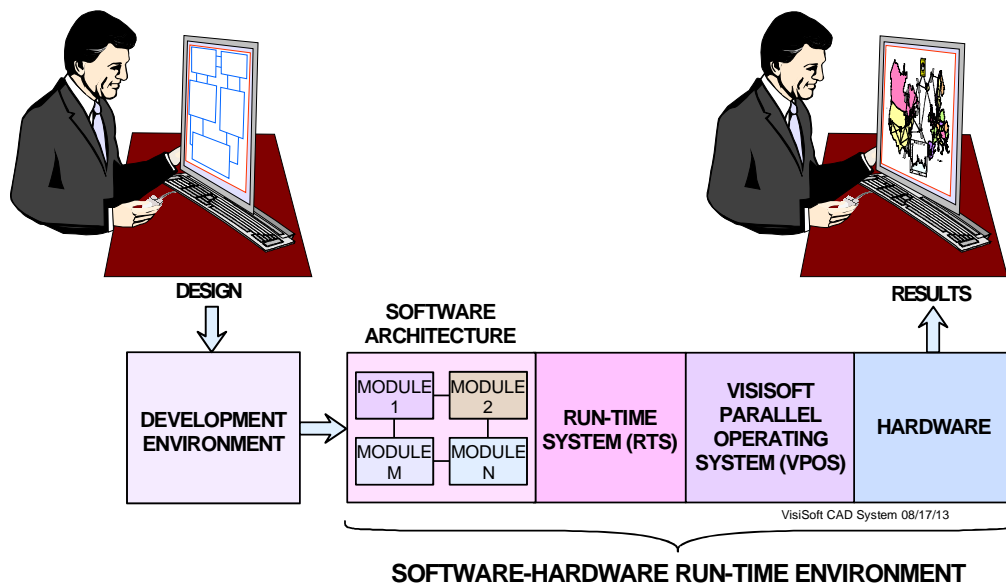


Figure 14-1. The Software-Hardware Run-Time Environment

One of the most important concepts already brought forth is that of using the development environment to derive key information about the application that can be used during run-time. Implementation of this concept is embodied in the development of a representative software architecture. This requires producing an effective mapping of the inherent parallelism of the application system into a software architecture of IND Modules.

Given that an optimized architecture has been produced in the development environment, the pertinent information it contains must be made available to the run-time environment. This requires that additional software be automatically generated in the development environment that is tailored to the particular application and hardware to create a very fast run-time environment. The software that is automatically created is the Run-Time System (RTS). It contains the required information about the application that is needed by both the special operating system (VPOS) and the hardware environment to maximize speed.

RUN-TIME ENVIRONMENT FUNCTIONS

When generating a parallel processing simulation or software architecture using VisiSoft, the designer must be aware of special Run-Time System functions that are provided automatically as described below.

- **Processor Allocation** - At the beginning of a simulation, IND modules must be assigned to a processor. The VPOS Module Management subsystem must know what processors are available for allocation to a module, what modules and instances may be assigned to processors, and then control the assignment of all processes in a module (instance) to the allocated processor. This information is provided by the RTS during initialization.
- **Thread Initiation** - The Schedule statement causes a process to be entered into the schedule. When popped off the schedule, that process initiates a thread. If that process calls other processes that in turn call other processes, they are all part of that thread. Threads may run on a single or parallel processor where they may run concurrently when in IND modules on separate processors. Threads may start other threads. At the completion of a thread, the next process is popped from the schedule.
- **Multi-Processor Scheduling** - Once modules are running on their assigned processors, threads can be scheduled to run on the same or different processors. If the application has a reasonable degree of inherent parallelism, most of the threads scheduled to run will reside on the same processor (each processor has its own schedule). Entries for processes scheduled to run on a different processor will be sent automatically to the scheduler residing on the assigned processor. These entries contain the process name, schedule time and priority, and up to six instance pointer names.
- **Module Migration** - If module migration is invoked, IND modules designated for migration may be moved during the course of the simulation.

HANDLING SCHEDULES AND CALLS AT RUN-TIME

This section is concerned with procedures for assignment of processes to processors. Note that these procedures deal only with processes. The assignment of resources to a particular level of memory hierarchy associated with a specified processor is addressed later. The user functions for CALLS and SCHEDULES are defined in Chapter 12. This section is concerned with how the Run-Time System (RTS) handles CALLS and SCHEDULES.

Figure 14-2 provides a top level view of key RTS functions for parallel processors. The yellow box in each processor represents all of the modules, threads, processes, and resources that make up the application simulation or software. The other boxes are part of the RTS (pink) or VPOS (maroon) that allocates and assigns processes to processors, and schedules threads to run on the various processors at the proper time. The next sections provide an overview of the procedure for assigning threads to processors, accounting for initialization and termination.

We note that Figure 14-2 shows multiple graphical workstations attached to the parallel processor via shared memory. This assumes that the workstations are tied into a server environment via graphics cards and are accessed via shared memory with the parallel processor.

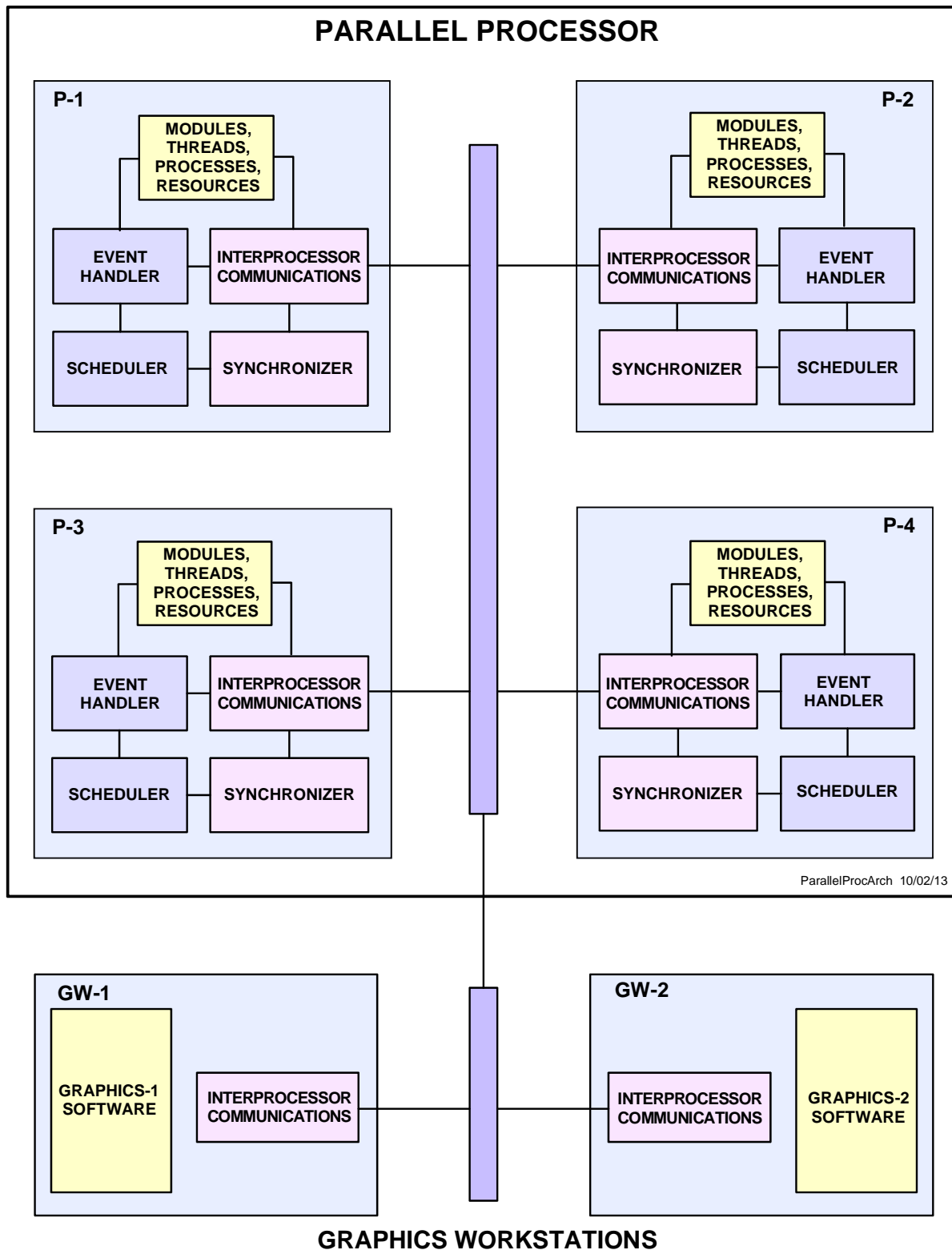


Figure 14-2. Top level GSS Run-Time System architecture for parallel processors.

OVERVIEW OF A SIMULATION/SOFTWARE RUN

When using parallel processors, it is convenient to separate initialization processes from those that do not participate in initialization. There are two types of initialization: system initialization and user defined simulation (application software) initialization. Simulation initialization may be run on a single processor. These are different as described below.

System Initialization

During system initialization, the RTS initializes itself to prepare to run a simulation (or software application). The RTS then proceeds to initiate simulation initialization (software application).

Simulation/Software Initialization

Application software initialization consists of two types. First is initialization of resource attributes that have been given INITIAL_VALUES. This is done when the module containing the resource is first assigned to a processor.

The second form of initialization is running initial threads that are designer specified as processes to be started in the DEFINITION, IDENTIFICATION, MODEL, or EVALUATION sections of the SIMULATION CONTROL SPECIFICATION. These lead processes are flagged in the architecture of the simulation using the MODIFY PROCESS panel shown below. These processes are then marked as START processes in the drawing of the architecture. In addition to selecting the section in which a process is to be started, the designer can select a priority from 1 to 99 so that processes may be started in a selected order within the specified section. One may also select the time units for scheduling, the instance pointer names, and a generic name (refer to Chapter 12).

Start Section & Priority	
Definition:	<input checked="" type="checkbox"/> 1
Identification:	<input type="checkbox"/> 5
Model:	<input type="checkbox"/> 5
Evaluation:	<input type="checkbox"/> 5

The Module Manager contained in VPOS takes the START process entries for the DEFINITION SECTION and sends them to the Scheduler (on P-1 in Figure 14-2) effectively scheduling these processes to run at time 0. These threads initialize the simulation and are started once, even when the multiple simulation facility is used.

Initialization Of Multiple Simulation Runs

When the initialization threads specified in the DEFINITION SECTION are complete, the Module Manager takes the START process entries for the IDENTIFICATION SECTION and sends them to the corresponding Scheduler to be run at time 0 in priority order. These threads are used to initialize each new simulation when multiple simulations are run, e.g., for Monte Carlo analysis. When each simulation completes, if there are more to be run, the threads in the IDENTIFICATION SECTION will be scheduled again to initialize the next simulation run.

MAIN SIMULATION (SOFTWARE) SYSTEM - MODEL (MODULE) SECTION

From here on in this chapter, simulation may imply software and model may imply module and vice versa. When initialization is complete so that the main simulation can run, the Module Manager takes the START process entries for the MODEL SECTION and sends them to the Scheduler. When all of the START processes have been scheduled, the Scheduler pops them off the Schedule to be run at the starting time in priority order. This starts the (next) simulation running. When a process is popped off the Schedule, it is given to the Module Manager. If the process resides on that processor, it is started. If it resides on another processor, the starting information is sent via the Synchronizer to the proper processor. All Inter-Processor Communications are handled by the IPC subsystem.

EVALUATION

After each simulation completes, the Module Manager takes the START process entries for the EVALUATION SECTION and sends them to the Scheduler. The threads that are started in this section are used to evaluate the results of each simulation, comparing them to all of the prior results if desired by the designer. These threads are all run on a single processor.

MODULE MANAGER

The Module Manager handles IND modules that have not yet been run. In the MODEL section, it must determine if the process starts a thread that may run concurrently as part of an IND module and if a processor has been allocated to that module or instance of a module. It is responsible for ensuring that all conditions are met before those threads can start. Conditions include the following:

- The starting process for the next thread is popped off the schedule.
- The IND module containing the selected thread must be assigned to a processor, so all processes and resources needed to support it are available on that processor.
- All submodules invoked by a thread must be assigned to the processor containing that thread. Communications with modules on another processor require standard interfaces to the IPC system.

CONTROLLING PARALLEL THREADS AT RUN-TIME

Figure 14-3 illustrates threads residing on multiple processors. Although not shown, all processors contain a VPOS Module Manager and Scheduler, and an RTS IPC Manager and Synchronizer. Some of these facilities were described in the prior section. They control the user application processes that all run as threads in Figure 14-3. The RTS and VPOS facilities that reside on each processor and control the application software also run as threads.

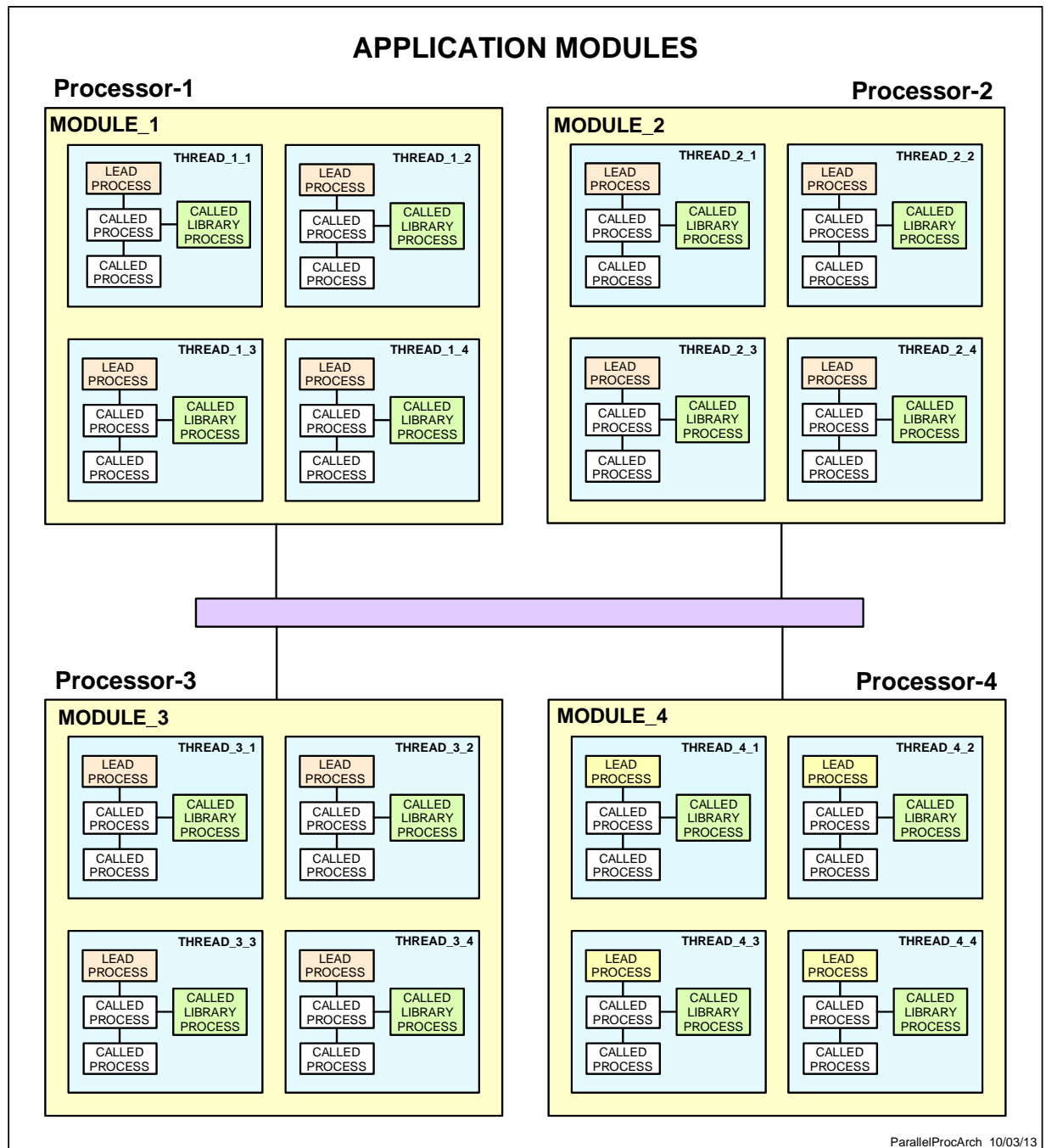


Figure 14-3. Illustration of threads residing on parallel processors.

MODEL (MODULE) SECTION Run Time Operations

The MODEL SECTION of the CONTROL SPECIFICATION corresponds to all models contained in the simulation or task architecture, including the instances that contain threads that are run as a result of those that are started in this section. When running on a parallel processor, the IND modules may be assigned to different processors to run concurrently. As specified by the application system design, these modules contain threads where each corresponds to a lead process that is started with a SCHEDULE statement. By virtue of the architectural design rules, all processes within an IND module must belong to one of the threads in that module. Threads may be SCHEDULED from anywhere in the simulation.

Threads may make CALLs to any process within the IND module in which they reside, or to UTILITY or LIBRARY modules that do not reside within that IND module, provided these UTILITY or LIBRARY modules are designated as *reentrant* (i.e., their operational outcomes rely only on resources shared with the calling process). The RTS ensures that copies of all utilities and libraries called from an independent module reside on that module's processor.

In large simulations using hundreds of processors, IP communications between different processors may incur significantly different memory boundary crossing delays. The mapping of IND modules into processors will determine these delays and affect overall speed. To maximize speed implies a physical assignment of IND modules to processors based upon a mapping of relative memory access delays between processors into connectivity between modules. One may think of this in terms of mapping the IND module connectivity footprint into an optimized matching processor connectivity footprint. Although one may consider an optimal mapping scheme, it may or may not provide substantial returns. As indicated in the prior chapter, depending upon the application, the connectivity between modules may be highly nonstationary, possibly changing an optimized mapping to poor. Migration benefits depend upon the time constants of such nonstationarities. Hardware designers have made strides to minimize these delays.

The Module Manager must map the designated IND modules or module instances into a set of available processors as the simulation starts to run. Since it controls which physical processors are allocated to IND modules, only it can optimize the allocation and assignment of processors to IND modules, including run-time module migration. We note that migration must be done on an IND module basis.

Module Manager - Model (Module) Section Functions

When an IND module is assigned to a processor, all processes and resources residing in that module must be loaded into the assigned processor, along with any utility or library modules required. Once this load procedure is complete, the module ID and pertinent residence information are transmitted to the local Module Managers for their use when a thread within that module is popped off their schedule.

When a process schedules or calls other processes in another IND module that does not reside on that processor, it must be checked to determine if that IND module has been assigned to a processor. If the module has been assigned, the process ID may be sent directly to the processor containing the module. If not, that process ID must be sent to the VPOS Module Manager for assignment and subsequent scheduling of the process.

Module Manager Rules

The following is a summary of the rules that apply to any process taken from the schedule and sent to the Module Manager. As used from here on, “module” may imply IND module instance.

If the process is contained in an IND module that is *not yet assigned* to a processor, the module is checked to determine if a processor has been *allocated* to that module (instance).

If not, a processor is allocated to the module.

When a processor is allocated, then the IND module is *assigned* to the processor. This implies that all processes and resources residing in that IND module are loaded into the allocated processor, along with any utility or library modules required but not yet assigned to that processor.

Once this load procedure is complete, the IND module ID and pertinent residence information are transmitted to all local Module Managers for their use when a thread within that module is popped off their schedule. The module is then considered assigned.

The schedule entry for that process is then sent to the Module Manager residing on the assigned processor for entry into the schedule.

Local Module Manager Controllers

When a process is popped off a local schedule and it resides within a module assigned to that processor, it is run on that processor when it falls within the ΔT_{\max} interval. If it resides in a module that has been assigned to another processor, then the schedule entry for that process is sent to the process controller residing on the assigned processor for entry into its schedule. If not assigned, it is sent to the Module Manager.

RUN-TIME SYSTEM PROCESSES VERSUS USER PROCESSES

Run-Time System (RTS) Processes

RTS processes work with VPOS to control the running of user processes and provide other functions to support simulations or software. These exist on all of the processors shown in Figure 14-3.

ENSURING CONSISTENCY AT RUN-TIME

In a single processor Von Neumann machine environment, processes cannot run concurrently - they must run sequentially. Therefore two processes cannot use the same resource at the same time. However, on a parallel processor, this case must be avoided. In the CAD environment described here, when processes residing on different processors are scheduled to run at the same time, the architecture environment shares the required information with the RTS to ensure that if resources are shared, they are updated in a synchronized manner. This generally implies that when processes run on a parallel processor, the results they produce are complete and consistent. In other words, given the same initial conditions on the resources connected to a process, the process will produce the same results each time.

When running a simulation on a parallel processor, processes residing in independent modules may run in parallel on different processors. When modules are running on the same processor, resources not shared with processes on another processor cannot be corrupted by unsynchronized updates. This is because processes on the same processor are scheduled in the same manner as if they were in a single processor simulation. However, processes running on different processors may be scheduled at the same or slightly different times - relative to each other - compared to when they are run on a single machine. This is because of the desire to minimize run-time speed when using a parallel processor.

As described in Chapter 6, to achieve higher efficiencies from parallel processors, one can allow the simulation clocks on different processors to drift apart by ΔT_{\max} , a parameter specified by the designer, without losing accuracy of simulation results. It has been documented and substantiated by experiment [114], that efficiency may increase substantially with ΔT_{\max} . It is up to the designer to mitigate the data corruption that may occur in IP resources as ΔT_{\max} increases from zero. This requires modeling interfaces between units to accurately represent the protocols, considering the variations that can occur in real physical situations. This is a necessary but not sufficient condition to ensure consistency.

To achieve higher efficiencies without losing accuracy, IP resources must adhere to a synchronization protocol. This requirement is described below. The IPC manager performs this function as part of the RTS.

There are two types of communications protocols needed to support fast processing of simulation modules at the interface between many processors. These are the following:

- **Asynchronous** - The latest copy of an IP resource (the one connected to the process that ran last) gets passed to the next process that needs it in another processor. Data coherency is guaranteed (independent of time).
- **Synchronous** - Everyone gets the latest copy from a single source, a one to many interface. This depends upon time synchronization required by models in the simulation (by design) to ensure data coherency.

The IPC protocol used in the RTS is the synchronous protocol because of the level of overhead involved in the asynchronous protocol. In addition, when two-way communications are required between processes, separate IP resources provide for full duplex communications. In the case of multiple computers, matching IP resources must use the same protocol.

Synchronous Communications Protocol

When the Synchronous Communications Protocol is used, only one process has write privileges to a given IP resource. In addition, copies are maintained at the receiving processes so that they are only updated before the receiving process is given control. If the process that writes the resource runs while the receiving process is running, the IP resource on the receiving side remains untouched.

To facilitate the use of IP resources, special facilities have been built into the architecture environment, development monitor, process translator, control specification translator, and RTS. These facilities provide for automatic recognition and handling of IP resources. They also provide for processing the control specification so that tables are built for the run-time environment that determines which processes share IP resources and what machines they should reside upon.

These facilities are designed to provide a clear speed advantage since (1) the CAD environment is tailored to a given platform; (2) the protocols used are generally transportable; and (3) speed is the predominate reason for using parallel processing. This is not an area where speed need be sacrificed for simplicity of the software.

When using full-duplex communications protocols, coherency is maintained implicitly. What must be controlled is synchronization. This is described in the following sections.

INTERPROCESSOR SCHEDULING AND SYNCHRONIZATION

To gain a speed multiplier close to N when using N processors, one must obtain a high processor utilization efficiency. Except for special problems, the processor utilization efficiency obtained today using current software techniques is typically around 10%. This implies a speed multiplier of 10 when using 100 processors.

As described in Chapter 6, for a simulation to obtain a high parallel processor utilization efficiency, many threads must be running concurrently on different processors. This is achieved when large IND modules are running concurrently on separate processors, with each module containing many threads. With sufficient inherent parallelism, most of these threads are scheduled by other threads within the same IND module. This implies that multiple threads are in the schedule on each processor, with most of them scheduled by other threads on the same processor.

By virtue of the independence properties of an IND module, these threads may run independently - with only a few sharing IP resources with the other processors, i.e., they are spatially independent with respect to the data. However, when running a simulation, they are not temporally independent. In this case, they must adhere to the simulation clock to run at their scheduled time.

SCHEDULING PROCESSES

Refer to Figure 14-4 for the following discussion. When using discrete event simulation, processes must run in time order based upon the simulation clock. When constrained to zero tolerance, i.e., $\Delta T_{\max} = 0$, the simulation clock may only advance when the next process in each of the schedules on all processors has a schedule time that is beyond the current clock time. This will allow the simulation clock to advance to the smallest increment of time of all of the scheduled times.

Using the CAD system described here, modelers use the SCHEDULE statement to cause processes to be placed in the overall simulation schedule to be run at a specified time in the future. Processes scheduled at the same time may be given a priority. If no priority is assigned, it is implied that the order does not matter, i.e., a valid result will occur if ordered randomly. We note that priorities are likely to be used only within an independent model (instance).

As described in Chapter 6, in simulations representing physical systems, valid results may occur even when processes scheduled at sufficiently small but different times are run out of order. This is because all physical systems have some degree of randomness. When comparing test data taken on physical systems, one can discover the value of ΔT_{\max} that will not affect validity. In a real system for example, if message A starts to come in before message B, but neither get processed until both are in, it does not matter which one comes in first. Or, if ten messages must come in before something happens, which ones get in under the wire may not matter because, in the real world, the results are valid either way. This is especially true when variations occur naturally, causing the finite distribution of the measurement vector.

This implies that, when a process is next in the schedule to be run, it must wait until the simulation clock advances to the schedule time of that process. To ensure that processes on different processors run in a sufficiently accurate order, the simulation clock on each processor must be synchronized so that it does not advance beyond a pre-specified tolerance ΔT_{\max} ahead of the clock on any other processor.

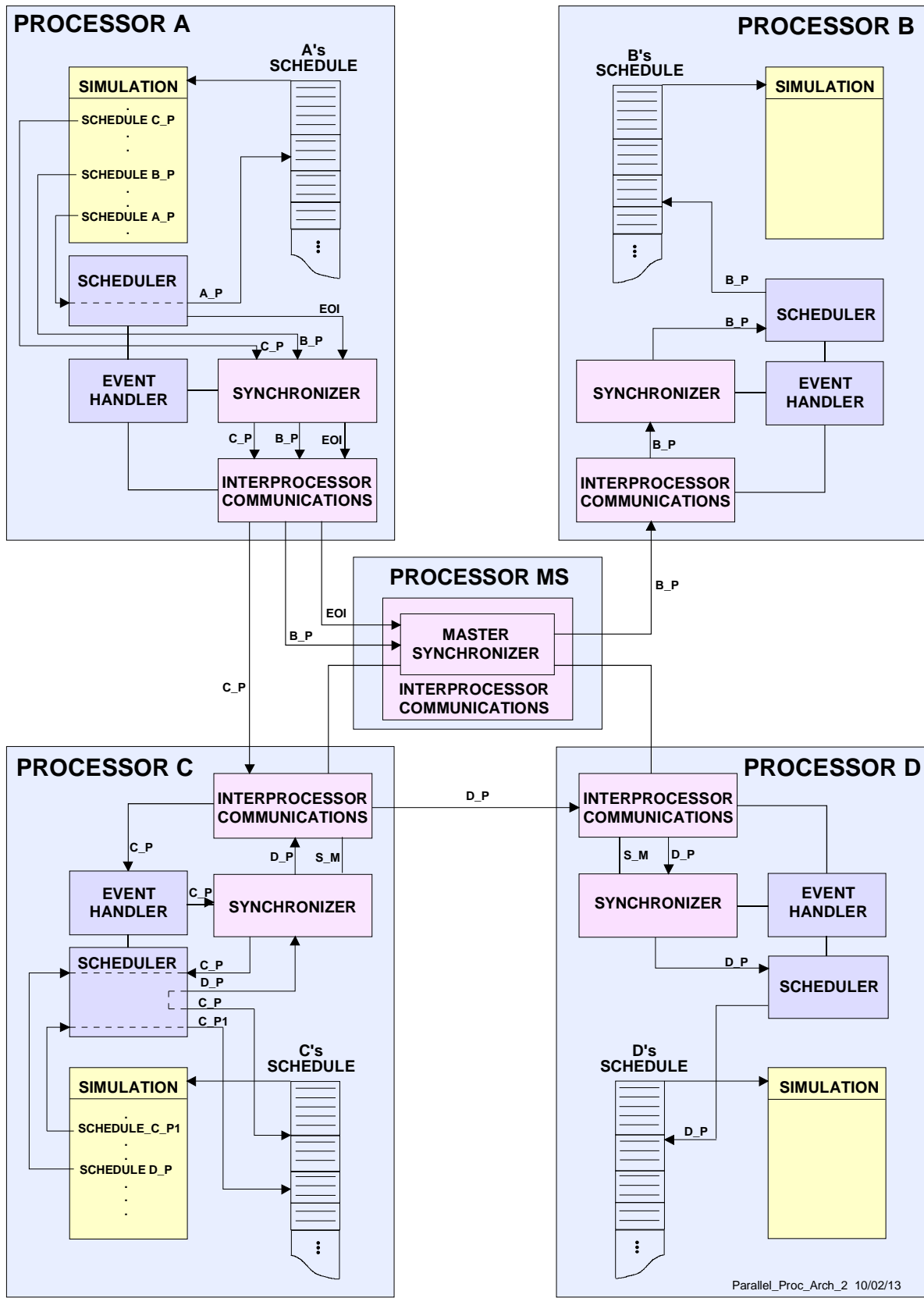


Figure 14-4. Cross-schedules for a four processor case.

Schedule Synchronization

When a scheduled process that is currently running on a given processor terminates, the next process is retrieved from the schedule. In the single processor case, the clock can simply be advanced at this time, T_c , and the process invoked. In the multi-processor case, if the clock advances beyond the $T_c + \Delta T_{\max}$ interval, then it must wait until all processors have reached the same condition. When this happens, a new $T_c + \Delta T_{\max}$ interval is set, and the checks are made again. Any processor with a process scheduled in the $T_c + \Delta T_{\max}$ interval can proceed to invoke that process. Others must wait for the proper interval.

When the simulation clock in any processor exceeds the $T_c + \Delta T_{\max}$ interval, a notification is sent to the master synchronizer containing the processor/simulation ID. In addition, all cross-schedule requests are sent to the master synchronizer before being sent to the processor containing the cross-scheduled process. This latter information is used to update the status maintained in the master scheduler regarding the number of processes (if any) to schedule beyond the $T_c + \Delta T_{\max}$ interval. For example, if processor A sends a signal to the master synchronizer that its simulation clock has exceeded the interval, but a cross-schedule is sent to that processor subsequently, it is not finished. However, the order of presentation will ensure that the master simulation clock will advance beyond the $T_c + \Delta T_{\max}$ interval only after the clock-time of the next process to be scheduled in every simulation is beyond the interval.

The Effect Of Allowed Variation - ΔT_{\max} - On Parallel Processing Efficiency

The objective of moving a single processor simulation to a parallel processor environment is to achieve run time multipliers as close as possible to N where N is the number of processors. This is achieved by increasing the processor utilization efficiency, which can be defined as the ratio of the time it takes to run a simulation on a single processor to that on a parallel processor, divided by the number of processors.

As indicated above, one way to accomplish this is to allow the schedule clocks on different processors to vary within a selected ΔT_{\max} . Various experiments have been performed using this technique, see [114]. To obtain valid results, this must be done while maintaining validity of the outcomes of the simulation. As shown from the theoretical curves in Figure 14-5, ΔT_{\max} plays a major role in processing efficiency. We stress that these curves assume that the simulation results remain valid. As shown in the figure, processor utilization efficiency varies differently for different simulations (A and B) achieving different levels with different values of ΔT_{\max} . As shown in the next section, it also varies with hardware architectures.

The shapes of these curves have been substantiated by experiment, see [114]. However, they are dependent upon simulation validity as a function of ΔT .

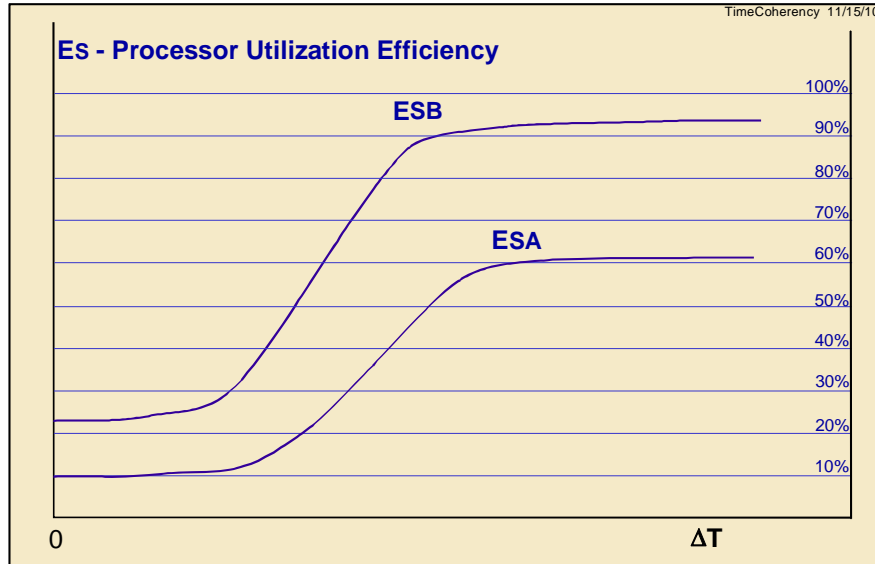


Figure 14-5. Parallel processor utilization efficiency as a function of ΔT .

The Effect Of Hardware Architectures On Parallel Processing Efficiency

We now consider the effect of different parallel processing architectures on the above curves. In this case only one of the curves for a given simulation architecture is selected while hardware architectures are varied. The theoretical curves are shown in Figure 14-6 and approximate those measured in an experiment, see [114]. The major variation in architecture was the speed of communications between processors, where Ep2 was approximately 10 times faster than Ep1.

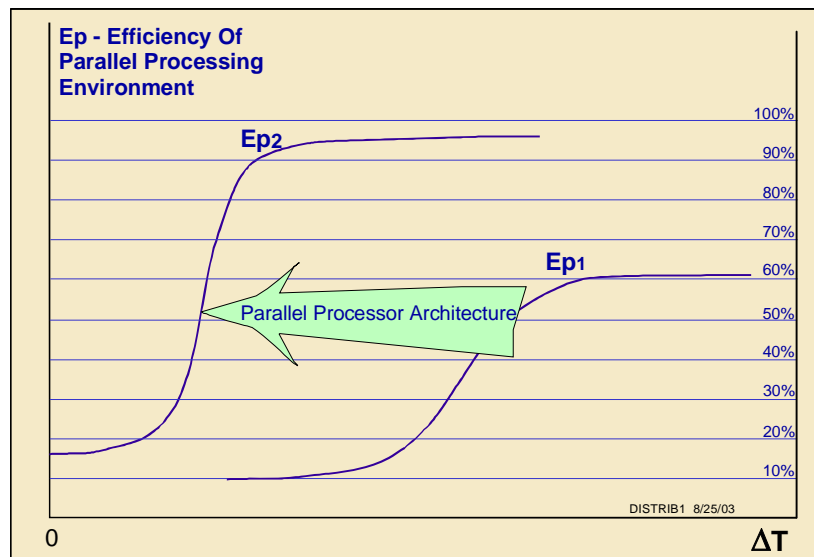


Figure 14-6. Parallel processor utilization efficiency as a function of hardware architectures.

Theoretically, differences in levels achieved may represent the different losses of efficiency incurred when:

- Transferring shared data from one processor to another
- Maintaining synchronization across processors
- Scheduling processes on different processors
- Idling due to imbalanced loading

Of the above bullets, the first three contribute latencies that reduce efficiency as they become a greater percentage of the overall processing time. Again, we must state that the validity of the curves shown in Figure 14-6 are dependent upon the validity of the simulation results. This dependency is addressed below.

The Effect Of Variation In ΔT On Simulation Validity

As described in Chapter 6, and repeated in Figure 14-7, the maximum value of ΔT that still maintains temporal coherency or synchronization, ΔT_{\max} , may be found for a given simulation. A ΔT_{\max} of zero may force the simulation to run with much reduced opportunity for increased processor efficiency. In fact, it may run slower on a parallel processor than on a single processor. As ΔT_{\max} is increased, more processing can take place in parallel before the simulations have to perform a time resynchronization. This increases efficiency.

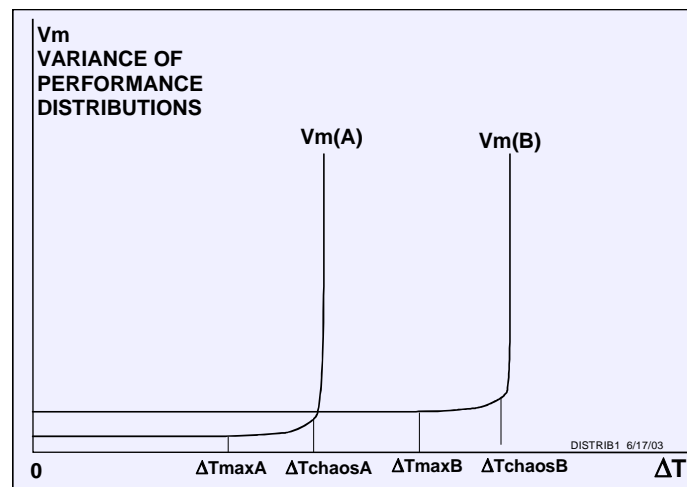


Figure 14-7. Cross-schedules for a four processor case.

As described in Chapter 6, an attempt to solving this problem was embedded in the Time Warp Operating System (TWOS), renamed SPEEDES, see Rieher [114]. This system allowed threads to run in advance of others (losing synchronization) with the idea that they could be “reprocessed” to resynchronize “out of sync” results. In partially independent cases, unscrambling the resulting chaotic states is virtually hopeless, and validity of results is clearly lost. The TWOS phenomenon was never clearly explained. Based upon the above analysis, the problem appears quite obvious.

For simulations of nonlinear systems, the variance of the distributions is typically unchanged until a break point, ΔT_{chaos} , at which point results become chaotic. If the systems being simulated have a synchronized component, i.e., events occur on a time synchronized basis, then this effect can be expected to occur as ΔT crosses the time synchronization point. Judgment can be used to back off to a valid point. In the communication system simulations used to test this approach, [114], the break point occurred at about 1 second. Backing off to $\Delta T_{\text{max}} = 0.8$ seconds was sufficient to obtain results that were virtually identical to those on a single processor or with $\Delta T_{\text{max}} = 0$. If changes are made to a simulation, ΔT_{max} must be revalidated.

Simulations of Time Domain Multi-Access (TDMA) radios may place a more stringent requirement on ΔT_{max} . For example, military networks use JTIDS terminals with a time slot of 7.8125 milliseconds. If the modeler schedules events within a time slot, a ΔT_{max} of 7.8125 milliseconds is likely to ensure validity of simulations that model message traffic to the time slot level. Depending upon requirements on the simulation, modeling to the JTIDS frame level may relax this requirement to more than 10 milliseconds (a frame covers 12 milliseconds). But the modeling approach itself may severely limit the validity of the simulation to investigate network performance when subject to rapid response requirements at the individual message level.

We note that validity as a function of ΔT_{max} is generally independent of the parallel processing environment. However, the actual curve will encounter variations resulting from the effects of random ordering of processes, producing a distribution of results caused by these random variations. This distribution should be within the simulation validity requirements.

Variations In ΔT On Parallel Processing Efficiency - Design Considerations

Given that the model and run-time software architectural approach takes full advantage of the inherent parallelism of a system, we must ensure that if we select a hardware architecture, it will meet the time and validity constraints. To illustrate the selection process, we use an example of run-time constraints where a 2 hour scenario must run in less than 6 minutes to achieve the desired goal. This implies a simulation-time to real-time ratio of 20. If the simulation runs 3 times slower than real time on a single processor, it must run with greater than 60% efficiency using 100 processors to achieve a speed up factor of 60.

Figure 14-8 illustrates two different hardware architectures, Ep1 and Ep2, as candidates to achieve the goal. In the case of Ep1, the efficiency remains at about 10% as we approach ΔT_{max} . This illustrates the effect of latency on achieving a feasible solution, i.e., meeting the time and validity constraints. Ep2 achieves 90% efficiency prior to reaching ΔT_{max} . For this example, Ep2 clearly exceeds the speed constraint, allowing one to reduce the number of processors used.

At this point we must note that this comparison assumed that the resulting speed up factor was based upon the parallel processor simulation being virtually the same as that used on the single processor version. When considering existing simulations built using current approaches, we must remember that they will likely run considerably slower just because of the language shortfalls - from a single processor standpoint - so that the speed multiplier may be much higher than illustrated in the Figure.

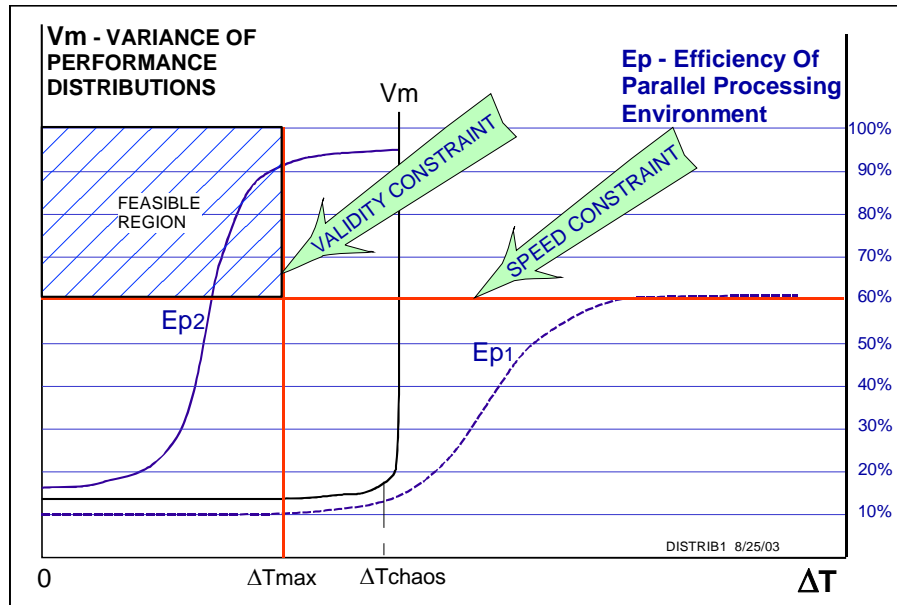


Figure 14-8. Overall considerations in selecting ΔT .

Synchronizer Design

When a process passes the interval test, i.e., it falls within the ΔT_{max} interval, it is checked to determine if it shares an IP resource. If it does, then an IP resource synchronization check is made. If that process shares an IP resource, then it automatically will have the latest copy of that resource. This is just one of the benefits of this CAD approach

The $T_c + \Delta T_{max}$ limit is determined by the master synchronizer after receiving the next schedule time from each processor and determining the earliest. After the current interval has completed, i.e., there are no more processes in any of the processors' schedules whose schedule times fall within the current interval, then the master synchronizer notifies all of the local synchronizers of the new T_c value, signaling the start of a new interval. The local synchronizers handle these decisions accounting for cross-processor schedules.

Rather than waiting until the interval is empty, each scheduler can provide the next-scheduled process time to the local synchronizer to pass to the master synchronizer. This information can be used to slide the $T_c + \Delta T_{max}$ interval forward to the next earliest synchronization time, T_c .

Master And Local Synchronizers

Figure 14-4 shows the master and local synchronizers. The master synchronizer tracks the earliest scheduled time of processes in the ΔT_{max} interval. It determines when the clock can be set to the next synchronization interval, and handles cross-schedules going both ways on processor A. The local synchronizers on the other processors report their earliest schedule time, to the master, and when they have reached the end of their ΔT_{max} intervals.

LANGUAGE TRANSLATOR INPUT

The control specification provides for a statement that specifies the ΔT_{\max} interval. The format for this statement is shown again below for convenience.

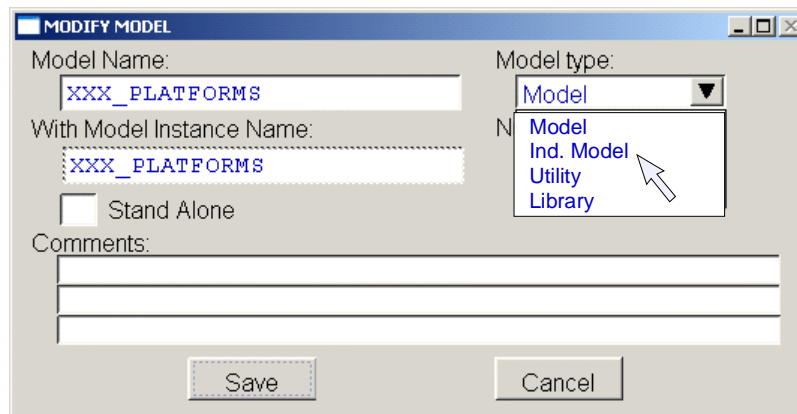
Format
<code>DELTA_TIME = $\Delta t_{\max_interval}$ [time_units]</code>

The time_units options are given in Appendix 4 of the GSS and VSE User's Reference Manuals, [67] and [150]. They range from PICOSECONDS to DAYS. If the time_units option is not used, time is assumed to be in seconds (the default). If a DELTA_TIME statement does not appear in the list, it is assumed to be 0 (the default).

When the parallel processor option is specified in the control specification, a schedule statement is translated such that code is generated that invokes the Module Manager and Synchronizer to initiate a schedule request to the processor containing that process.

DESIGNATING MODULES FOR ALLOCATION

It is the designer's responsibility to designate those independent modules that can be assigned to different processors by the run-time monitor. To do this, one must use the MODIFY MODULE panel and select Ind. Model as shown below.



Comparison To Current Approaches

Current approaches to parallel processing only generate the object modules for the source code written by the programmer. They do not generate an RTS. The RTS is a separate software subsystem that is based upon the architectural information generated by the designer. This includes the facilities that provide the following:

- Rapid synchronization of IND Modules that exchange information;
- Inter-Processor Communications (IPC), a system that supports the IP Resources;
- Information needed by VPOS to perform optimal allocation of hardware resources.

Because current approaches do not generate an RTS in the development environment, existing systems provide little information to the OS, expecting it to figure out how to break up individual routines using methods of tiling. Tiling essentially breaks out the code inside loops to place small amounts of instructions on separate processors. Otherwise, these systems provide special instructions to the user for dealing with the run-time issues.

Another important ingredient of the RTS is thread synchronization. By ensuring that threads reside within IND Modules, threads must run on a single processor, ensuring their temporal independence. Threads may initiate other threads on any processor, but these are automatically synchronized by the RTS.

The combination of spatial independence, IP Resources and synchronization of threads ensures consistency of results across processors. The need for cache coherency is eliminated, making chip space available for more memory - the major driver for increasing speed.

The authors are not aware of any other integrated approach that helps the user to map inherent parallelism into IND Modules using a graphical architectural framework. We know of no systems providing information exchanges regarding the design of the architecture or generation of run-time code to take maximum advantage of the hardware environment. In fact, we know of no systems that provide for software architecture.

We also note that the combination of the RTS and VPOS allows the end-user to vary the number of processors used simply by changing the Control Specification. This makes it easy to compare running times on a single processor to those on different numbers of processors. One can then plot a curve of different numbers of processors to determine the best hardware configuration for meeting the speed requirements of a given application.

CHAPTER 15

VPOS - A PARALLEL PROCESSOR OS

This chapter describes the top level design of the VisiSoft Parallel OS (VPOS). We start by assuming that the Software Architecture and Run-Time System (RTS) shown in Figure 15-1 have been produced using the VisiSoft approach described in prior chapters. In the next chapter we will discuss the theoretical requirements for design of the hardware. We make assumptions with regard to hardware design here with three qualifications. First, we assume existing chip designs exist that are relatively simple and follow the typical processor architectures. Second, we assume that VPOS can be tailored to different chip designs relatively easily. Third, we assume that hardware architects will understand the theory presented here and will heed the suggestions made in the next chapter with regard to future parallel processor designs.

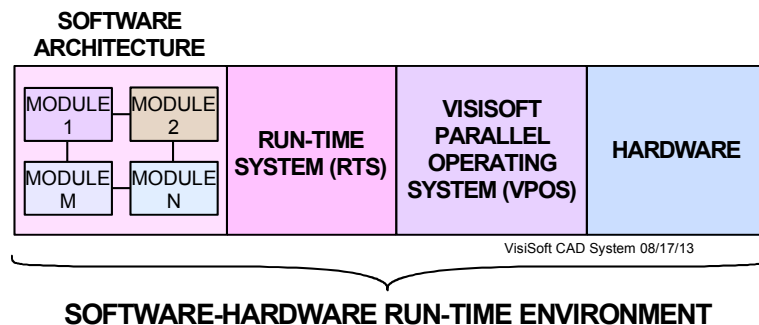


Figure 15-1. The Software-Hardware Run-Time Environment

Focus On Parallel Processors

VPOS is not intended to be a general purpose operating system. It has been designed to operate on a shared memory basis with a server environment as described in Chapter 3, and illustrated in Figure 3-6. This implies that information exchanges between the parallel processors, including those to the server, are done using shared internal memory - with no direct external device channels except for short distance communications. When applications interface with external devices, e.g., keyboards, mice, screens, printers, graphics cards, disks, long distance communication channels, etc, these are through special interfaces with processors in the “server” environment. Therefore, VPOS contains no device drivers and no secondary storage file management facilities. Instead, its facilities are focused on optimal management of a specified subset of fast parallel processors assigned to a specified task.

In the case of a Personal Computer operating under a standard PC type OS, e.g., Windows, Linux, UNIX, or others (we use WINUX to designate these), the computer system takes on the type of hardware architecture similar to that represented in Figure 15-2. Whereas Figure 3-6 shows the buses used for shared memory channels, that level of detail - although obviously necessary - is not represented in Figure 15-2. We note that a WINUX type OS will control the processors (green) allocated to it, while VPOS controls the other processors (blue). We also note that a part of VPOS resides on each of these processors, while the top level subset of the VPOS system resides on a specified processor.

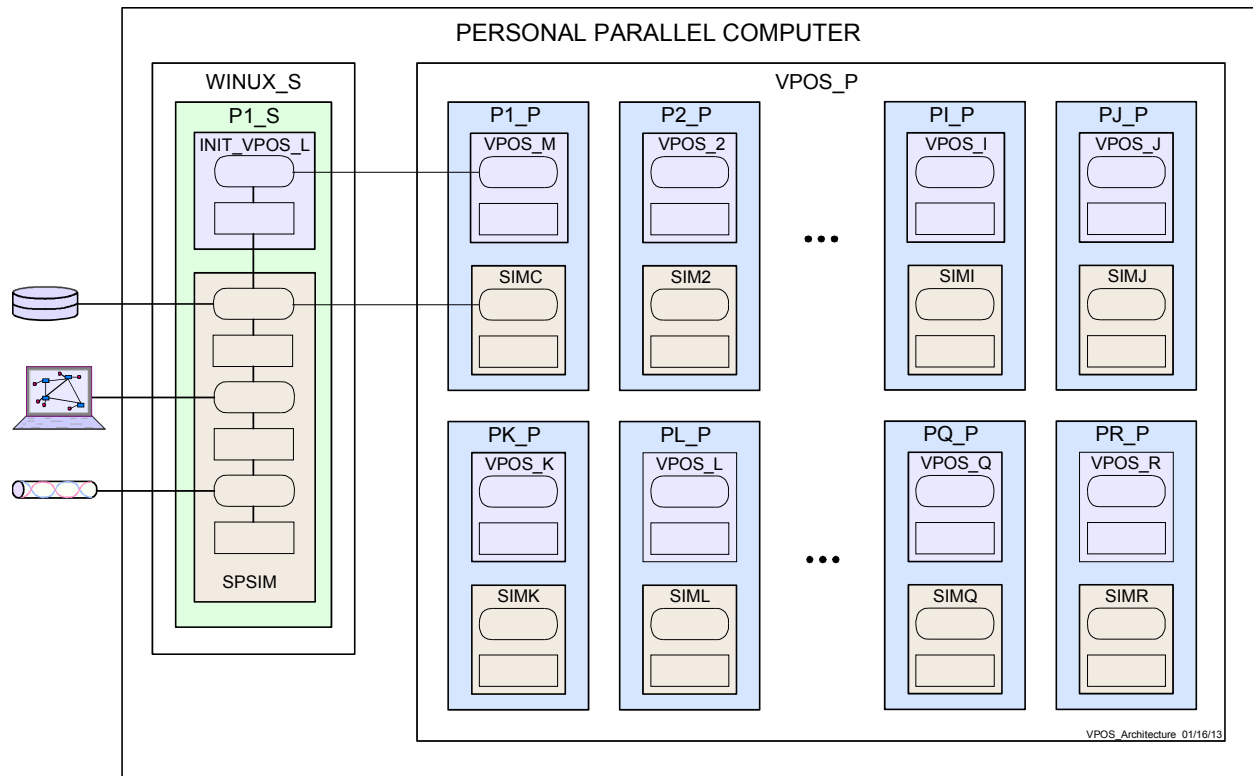


Figure 15-2. A Parallel Personal Computer with WINUX and VPOS running concurrently.

In the case of the Parallel PC (PPC), one may use a total of 36 processors. One or two of these processors may be used to house the Windows or Linux OS (shown as WINUX). The WINUX OS interfaces with the external device drivers. The remaining processors are used to support parallel processing of one or more tasks that must run fast. Because of the speed multipliers achieved by the VisiSoft Run-Time System, it is likely that a task or simulation requiring 3000 processors on an HPC may be run at least as fast on a PPC with 36 processors.

As indicated in Chapter 14, VPOS allows the end-user to vary the number of processors used simply by changing the Control Specification. This makes it easy to compare running times on a single processor to those on different numbers of processors. One can then plot a curve to determine the best hardware configuration for meeting the speed requirements of a given application.

OVERVIEW OF THE VPOS ARCHITECTURE

Being able to control the design of an operating system using engineering drawings is likely one of the best examples of the use of VisiSoft. Although the device drivers make up a major part of most operating systems, they are inherently independent and are generally written by the device manufacturers to be incorporated into the OS. The major parts of a multi-tasking virtual memory OS for a parallel processor are illustrated in the top level drawing of VPOS illustrated in Figure 15-3.

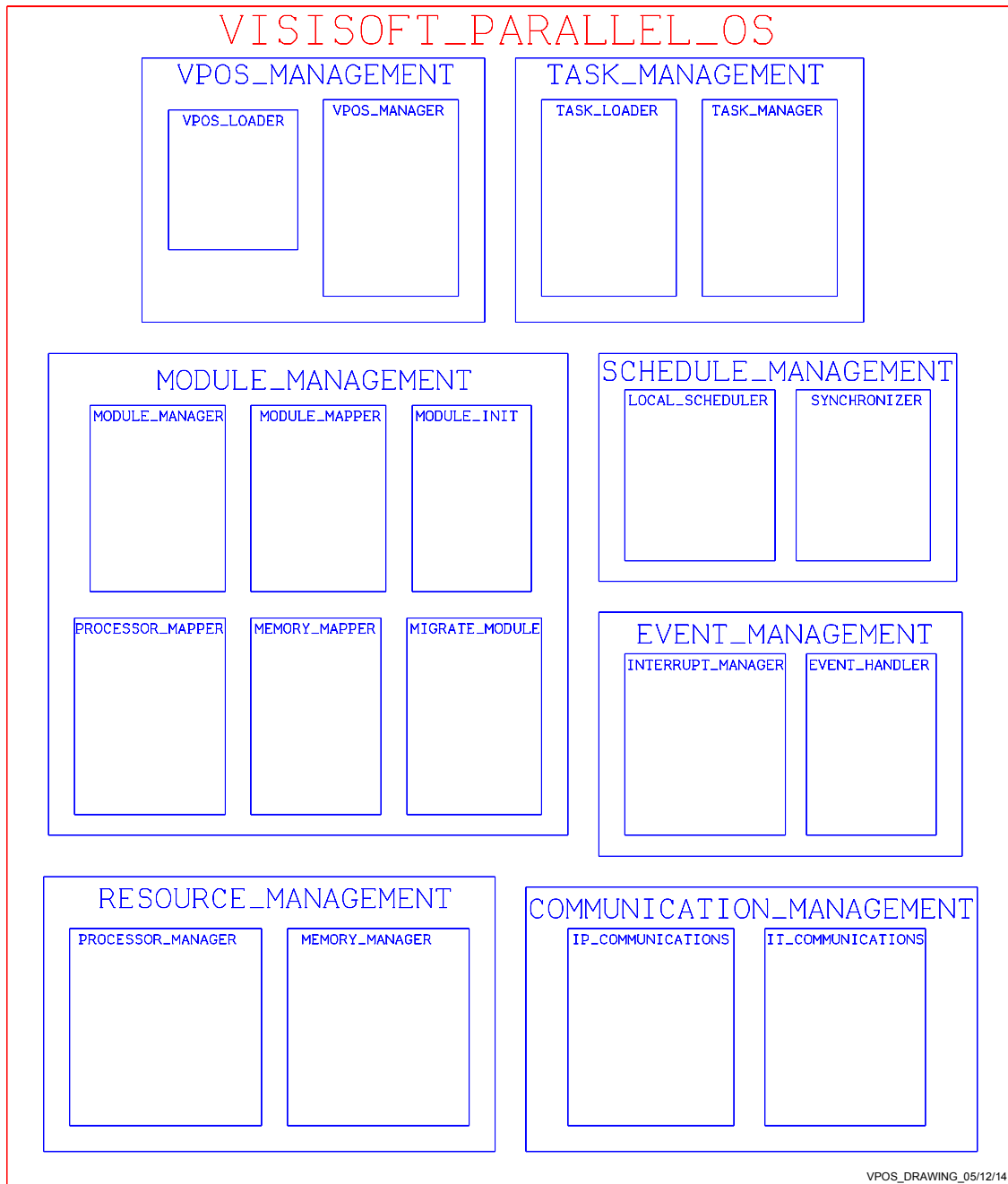


Figure 15-3. A simplified overview of the VPOS architecture.

Although not described here, there are two ways to operate VPOS. The one not discussed here provides a simplified installation on a PPC under the WINUX OS. This installation does not use the full-up VPOS shown in Figure 15-3. However, from a user standpoint, it appears substantially the same. Depending upon the application, it may run slower than that described here because all of the facilities of VPOS shown in Figure 15-3 are not implemented in that version.

INITIATING PARALLEL PROCESSOR TASKS

When a parallel processor task is spawned by a server task, control is passed to the VPOS to initiate the task. In the case of IND modules, the following is implied:

- The parallel task has been designed with multiple IND modules that are capable of running concurrently on separate parallel processors.
- Object code has been produced along with special facilities in the Run-Time System (RTS) to support automatic synchronization of threads and IP Communications (IPC) for IP Resources contained within IND modules that may run concurrently.
- Each IND module is linked separately with special library modules that interface to the RTS and VPOS, as well as with the Utility and Library modules required by the application software in that IND module.
- Before the spawned task is loaded, it must be allocated with the required number of processors.
- IND modules must be assigned to separate processors to maximize PUE while meeting speed constraints. These modules are loaded onto the assigned processors.
- When IND modules are loaded onto their assigned processors, they are linked to the RTS and VPOS, and their resources are automatically initialized where applicable.

This requires special linking and loading facilities to support the RTS and VPOS. These facilities must be compatible with the WINUX OS as well as VPOS.

VPOS MANAGEMENT FACILITIES

The following sections provide an overview of the subsystems making up the VPOS management facilities.

VPOS MANAGEMENT

This subsystem is responsible for loading the VPOS itself onto the parallel processors allocated to VPOS. It interfaces with the spawning module on the WINUX OS to load the initiating part of VPOS onto the assigned VPOS processor. It then loads the remaining part of VPOS onto that processor. It also loads those portions of VPOS onto the other processors as necessary to support a specific task.

TASK MANAGEMENT

This subsystem interfaces with the spawning task running under WINUX to initiate running an application task on one or more parallel processors under VPOS. This includes setting up interfaces to the device handling facilities needed to support the parallel task.

MODULE MANAGEMENT

This subsystem manages the IND Modules that form a parallel processing task running under VPOS. If the user has specified which modules are to run on what processors, then the subsystem allocates and assigns the desired processors to the task. If processors are not assigned to modules by the user, this subsystem allocates and assigns available processors to the modules based on built-in optimization parameters. It also initiates the initialization of modules. If desired, module management can migrate modules to different processors while the task is running to minimize communications delays and unbalanced loading.

SCHEDULE MANAGEMENT

When a task is running IND Modules on multiple processors, multiple threads may be scheduled to run at a future time or NOW, implying after the current thread completes on that processor. To avoid bottlenecks, a scheduler exists on each processor. Depending upon the number of processes that may be scheduled at a given instance in time, the schedulers must be able to support the maximum number.

Running processes may schedule threads in different IND Modules that are running on other processors. This requires synchronization among the schedulers on these processors. The synchronization system is produced by the development environment and becomes part of the RTS which interfaces with VPOS.

There are two levels of synchronization, one within the ΔT_{\max} interval described in Chapter 6, and one outside. When the schedule time of a thread falls within the interval, it must be synchronized to ensure that the interval will not end before it is run. If it is scheduled outside the interval, then the level of synchronization must be handled before the start of the next interval.

EVENT MANAGEMENT

There are two types of events that must be handled. One is an externally generated event, e.g., coming from the Run-Time Graphics (RTG) interface or some other hardware request. These are generally considered Interrupts and are fielded by the Interrupt Handler. The other type is initiated by an internal statement and is handled by the Event Handler. Both types of events must be handled within a ΔT time step to ensure proper synchronization with external processes.

RESOURCE MANAGEMENT

The resource manager is concerned with managing the use of machine (processor and memory) resources. This is a careful management function in that only one process may control a resource at one time. This implies that the assignment of control over these resources must be done in an explicit manner, with no question about who has control. The major resources of interest are those of the processors and the memory. The allocation and assignment of individual resources must be under the explicit control of the WINUX OS or the VPOS.

Processor Management

There are many functions of a parallel processing OS that are critical to achieving high processor efficiencies. One of the most important is processor management. Mapping IND Modules onto physical processors is affected by the following factors.

- IND Modules that communicate most and significantly affect the total time to run are best mapped onto adjacent processors with minimal time delays.
- IND Modules that do not communicate need not be close.
- IND Modules that change their need to communicate are best remapped based upon their needs if the time-constants permit (the non-stationary case).

The above factors are determined by the Application Space Architecture (ASA). To minimize the delay times between processor chips, boards, and trays, the hardware architecture must follow the ASA so that communications are always between adjacent units. This requires special optimization techniques for spatial mapping of the ASA onto the hardware architecture, as well as optimized hardware designs.

Memory Management

Effective mapping of software onto hardware implies mapping a software architecture into a hardware architecture. Optimizing speed requires some knowledge of hardware architecture. Mapping instruction and data spaces into physical memory is a prediction problem. Predictions are determined by conditioned probabilities. Prediction accuracy depends upon the information used to represent the conditions. The approach that has the most information and makes best use of it will produce the most accurate prediction.

Most Operating Systems provide a facility to do the software to hardware mapping automatically. Typically they do not have sufficient information about a particular application to do this effectively, and waste time doing it wrong.

The mapping process can be done during as well as before run-time. If the OS does not have the proper information during run-time, reallocation of resources can waste time instead of saving it. With a software architecture optimized for mapping into hardware, and a run-time facility that provides that architectural information to a tailored OS, run-time movement can be minimized if not eliminated.

Mapping IND Modules onto processors requires mapping them into physical memory adjacent to the processors. To accomplish this, memory mapping must include maps to processors as well as to real memory locations and identify memory shared between processors. It must determine the starting address of the first instruction for each processor. A simple example of mapping estimates of delays between RAM on different boards is shown in Figure 15-4.

In Chapter 16, Table 16-1 illustrates the huge differences in memory boundary crossing delays. These delays affect decisions by the operating system to move both instructions and data around to avoid these delays. Much time is saved when these swapping and paging delays are minimized if not eliminated. This leads to the different application types described below.

MEMORY ACCESS DELAYS												
RAM ON BOARD	1	2	3	4	5	6	7	8	9	
1		D12	D13	D14	D15	D16	D17	D18	D19	
2	D21		D23	D24	D25	D26	D27	D28	D29			
3	D31	D32		D34	D35	D36	D37	D38	D39			
4	D41	D42	D43	...								
5	D51	D52	D53		...							
6	D61	D62	D63			...						
7	D71	D72	D73				...					
8	D81	D82	D83					...				
9	D91	D92	D93						...			
...	
...	

MemoryMap 01/03/11

Figure 15-4. Illustration of a table of on board memory hierarchy delays.

If all of the code and data of an independent module fit into local chip memory (level 1 or level 2 cache), swapping and paging are unnecessary and speed is maximized. This implies that (1) they fit; and (2) they are not going to move (stationary connectivity). Alternatively, if they do not fit, but the statistics are still highly stationary, time spent swapping and paging will still be insignificant. This affects the trade-offs between memory size and special hardware algorithms for swapping and paging that use chip space. With enough memory in each of the areas in the memory hierarchy, swapping and paging time will be insignificant.

Stationary Applications

Stationarity implies that the same modules communicate with each other throughout the run. Given a mapping of software architecture onto a parallel processor hardware architecture that minimizes the memory boundary crossing delays, swapping and paging are unnecessary. Such a mapping is best left in tact, else time is wasted by the OS trying to find a better one.

Given that an optimal mapping is achieved, i.e., one that minimizes run-time, the OS must be notified not to change it. Else, if the OS tries to improve the mapping, it may start a random walk trying to find a better mapping, losing the starting point along the way. To benefit from good mappings, one must be able to flag the OS to leave the mapping stationary.

Tests For Optimal Mappings

VPOS contains data collection facilities that produce measures of useful and idle time for every ΔT_{max} window. VisiSoft also contains facilities that map these results graphically, on a deterministic or statistical basis. As shown in Chapter 18, these measures may be used to vary the mapping of independent modules onto processors to minimize the number of processors used while meeting the run time constraints.

Stationarity May Depend Upon Scenarios

A particular software application may be stationary under one set of input scenarios and nonstationary under another set. This is illustrated in Chapter18, where application platforms of a given type are mapped into IND modules. Platforms of one type may communicate with platforms of a different type as well as with each other depending upon their connectivity.

In addition, platforms may move to different areas where they no longer communicate with those with whom they were previously connected. Instead, they may now be connected to new platforms in the new area, and must be able to communicate with them. Because the modules represent instances of the physical platforms, and because they change their connectivity, they are now sharing memory with different modules.

Nonstationary Applications

To support scenarios that render architectures nonstationary, the architectures illustrated in Chapter18 may be redesigned. This typically involves grouping previously independent modules into a higher level independent module and expanding the number of resulting modules. This creates an abstraction that requires the platform submodules within an independent module to take on properties of another platform within a distant independent module. This requires moving the explicit data properties from one abstract module in a given IND module to another abstract module in a distant IND module.

The time constants between such changes are generally sufficiently large to limit the number of moves within the module design. To do this effectively, these multiple moves must be made within the software design based on knowledge of the multiple subsets of data. However, this abstract approach makes the module design much more complex. Although faster than the multiple OS level moves, they will take considerable time compared to a single IND module move.

When looking at the resulting expanded modules, one sees abstract copies of all of the submodules on each processor. With sufficient memory, this is not a problem. However, given the design of the IND modules, and the fact that VPOS knows the mapping of IND modules to the hardware architecture, it is relatively easy for VPOS to track transfers of memory among IND modules and remap the allocation of these modules to processors to minimize the memory boundary crossing delays. These transfers can be minimized and done together. Transferring an IND module to a different processor can be performed by VPOS in a few moves, so multiple modules may be moved to different processors very fast.

COMMUNICATIONS MANAGEMENT

We are concerned with two types of communications: those between tasks (Inter-Task) and those between processors (Inter-Processor). Those between tasks may also be between different processors. In a single task with IND Modules running on different processors, some of these IND Modules may also communicate with different tasks. Setting up IT or IP resources is a simple mouse click in VisiSoft. All of the facilities needed to support these resources are automatically generated as part of the RTS, making these facilities extremely easy to use.

VPOS - FOCUSED ON PARALLEL PROCESSING

What VPOS Does Not Do

VPOS is aimed at optimizing and managing the hardware space for parallel processing. When using parallel processors to gain speed, it does not make sense to inject large time-consuming tasks that bog down the parallel processor when they can be done outside the tightly coupled parallel processor boundary. This seems obvious, but is not generally followed in current OS designs. The only interfaces needed for a fast parallel processor environment are cache or other RAM. Let the server side of the farm support everything else, e.g., the following:

Keyboards	DVDs
Mice	Hard Drives
Screens	Communication Channels
Printers	Memory Sticks,
CDs	etc.

Use of these devices from the parallel processor can generally be treated using shared memory with standard simplex or full duplex protocols. There is no need for DMA channels, fast graphics cards, etc. These functions take significant amounts of overhead, obstructing the ability to gain speed, and can be dealt with *independently*.

Current approaches to parallel processing are similar to Blind Man's Bluff, leaving the ability to detect inherent parallelism to the operating system. The classic approach uses Tiling to break up DO/FOR loops. The classic example is matrix inversion. But fast matrix inversion is best done using the symbolic solution approach; there is no looping, and no inherent parallelism. However, this requires detailed knowledge of the application, something held by only subject area experts. Programmers generally do not have the requisite background to understand the application in depth, and particularly the real inherent parallelism.

What VPOS Does

To achieve maximum useful processing overlap, VPOS minimizes the time spent waiting for data. This is achieved using copies of memory and proper communications protocols. To minimize memory boundary crossing delays, VPOS takes in the RTS architectural information to optimize the physical mapping of IND modules onto processors using knowledge of the architectural connectivity of IND modules. These are just some of the features of VPOS.

CHAPTER 16

IMPROVING PARALLEL PROCESSOR HARDWARE DESIGN

This chapter examines approaches to parallel processor hardware design that take advantage of the Application Space Architecture (ASA). It is concerned specifically with the logical, electronic, and mechanical design of computers that can maximize the speed of applications using a large number of processors. The design objective is to minimize the number of processors required to meet a specified application run time constraint by taking advantage of the ASA. The applications of concern are those that require multipliers of N times single processor speeds where N may be multiple orders of magnitude. Using the ASA approach, one can expect to take an application currently requiring many thousands of processors and reduce that number by 2 to 4 orders of magnitude.

This will require design of the application software in VisiSoft to take maximum advantage of the inherent parallelism in the system - putting that parallelism into IND Modules. Using the VisiSoft architecture to rebuild an existing application should be relatively easy. More importantly, significant future expansion is greatly simplified. Given that we meet these goals, the economic savings in terms of time, money, power, floor space, and air conditioning are generally immense compared to the application redesign investment.

As in Chapter 15, we assume that the Software Architecture and Run-Time System (RTS) shown in Figure 15-1 have been produced using the VisiSoft approach described in the previous chapters. Second, we assume that VPOS can be tailored to different hardware designs relatively easily. Third, we assume that chip designers will understand the theory presented here and will make suggestions on improving the approach with regard to future hardware designs.

Chapter 15 described how simulation can be used to optimize the various VPOS management algorithms with VisiSoft. This is a highly nonlinear problem where improvements in the hardware allocation algorithms, using parameters or different schemes, are difficult to evaluate without simulations of representative applications. In addition, VisiSoft has built-in nonlinear optimization facilities that have evolved over many years of solving similar problems. These same simulation and optimization facilities apply directly to the design of hardware architectures. In particular, we are concerned with optimal use of chip space, design of the size and placement of memory hierarchies, and the use of direct memory transfers between adjacent boxes.

Part of the solution to this problem is provided by the VisiSoft optimization facility used to design the VPOS memory management algorithms. For example, the VPOS design allows the memory management algorithms to use variable parameters to represent the delays of a given hardware design. This can be used to produce parametric solutions that yield hardware design parameters. One can then perform parametric analysis to determine the best tradeoffs when allocating chip space and designing the physical layout of boards, trays, boxes and racks.

Clearly there are different types of applications that must be addressed, and these will present different loads to the hardware design. These different applications may be simulated in terms of the numbers and types of IND Modules they present to the system. Examples of these differences are clearly observed when running simulations that illustrate differences in IND Modules that relate directly to Figures 6-2 through 6-6, reflecting the inherent parallelism in the applications and the PUE. This is illustrated in experiments described in Chapter 18.

DEALING WITH PARALLEL PROCESSOR ARCHITECTURES

Computer hardware architecture was driven by clocked operation speed until the recent clock rate barrier was hit. When computer clock rates were doubling about every 18 months, application software speeds increased at the same rate without any software changes. Since the leveling off of clock rates, buying a new processor no longer achieves such speed improvements.

Parallel processors now dominate computer design. Chip manufacturers cite the number of processors (cores) on a chip, implying that this effectively multiplies the speed of the computer. An example was described in Chapter 3, and shown in Figure 3-3. The problem is that very few representative applications measure speed on a scientific basis. What is obvious is the need to uncover and sort out the facts.

One reason for the lack of measurements is the way software is built for single versus parallel processors. Clearly application software must be tailored for parallel processors to reap the full benefits. Even then, it is often tailored for a specific number of processors. With current software technology, changing software is time-consuming and rewrites are costly - even for a single processor. This inhibits comparison of single processor versus parallel processor speeds. For many applications, current software approaches make it difficult to compare speeds using different numbers of parallel processors.

Without measurements that generate good data, hardware designers have difficulty making design decisions. Instead they are driven by people lacking in-depth experience in both hardware and software. Even the ideas are based on questionable requirements. In addition, many HPC hardware designers are driven by buyers who are measuring how many processors may be strung together in a large parallel processor. This implies that the more processors one connects, the more speed one can gain. This can be deceiving from at least two standpoints. First, some new architectures are moving toward nano-processors, where each processor is less capable and has a much smaller amount of cache memory directly available to it. Second is the drive to Exascale computing. The idea is that speed is proportional to the number of processors or flops, e.g., in embarrassingly parallel applications. Even at smaller levels, this has been shown to be false.

The following important observations are taken from: *A Comparative Study of Parallel Sort Algorithms*, by Davide Pasetto & Albert Akhriev, IBM Dublin Research Lab, Dublin, Ireland.

Modern SMP architectures are evolving towards a common design: a number of cores connected to shared main memory via hierarchy of caches with increasing size and decreasing performance. The difference in latency and bandwidth between the cache memory and the main memory can be so high that algorithms considered very efficient from a theoretical point of view could have poor performance in practice. It is well known that an efficient implementation of any algorithm cannot ignore the underlying hardware architecture and this is even more important when designing a parallel implementation. Therefore, some basic knowledge about cache / memory interaction should be embedded in the algorithm structure. When the implementation is meant to run on a parallel system, the effects of cache coherence protocols make things even more complex: data structures and algorithms must be designed to avoid false sharing and unnecessary cache coherent operations.

One may argue that these observations depend upon the application. For example, certain “fine grain” applications (covered in a following section) are claimed to be well suited to nano-processors. The prominent example of such an application is a one-direction passing of independent data streams through a sequence of processors, e.g., pipelined graphical processing. But for main stream applications, we know of no parallel processor experiments and documented results to refute the observations. Most applications must deal with the physical realities of distance-to-memory or distance between processors.

One can show theoretically that speed may be increased by grouping the processing performed by many processors onto a single processor (an example is provided below). When one compares all of the overhead functions and distance factors that result from using 3000 processors instead of 30 in a single PC, it is clear that independent experiments and measurements are needed to support hardware design approaches. It is time to rethink how software architectures, embodied in ASA, affect hardware architecture, and vice-versa.

MEMORY BOUNDARY CROSSING DELAYS

To illustrate the effects of hardware architecture, Table 16-1 provides examples of possible ranges of time delays for memory segment transfers between different processors on a parallel computer. It also provides associated ranges on memory sizes and numbers of processors. The ranges are wide to allow for different designs and technologies. Access times within a processor (level 1 cache) or chip (level 2 cache) may take 0.2 to 2 nanoseconds. This may rise by a factor of 5 or more between chips on the same board (L3 cache). When going between boards, this may rise by another factor of 5, depending on the design. Similar increases in delay occur due to memory transfers between trays and racks. It must be emphasized that the numbers in the table clearly depend upon design of the hardware architecture and the electronic circuitry used to implement that architecture. We also note that these numbers have been improving with time and may continue to do so for the foreseeable future but the relative delays will likely remain.

Table 16-1. Processor and memory - sizes and delays.

DELAYS, MEMORY SIZE, & PROCESSOR COUNT				
Position	Time Delays (Nanosecs)	Memory Type	Memory Size (Bytes)	Number of Processors
With Processors	0.2 - 2.0	L1	1 - 4 M	1
Within chips	1.0 - 10	L2	16 - 128 M	4 - 18
Within Boards	5.0 - 50	L3	48 - 1024 G	24 - 72
Within Trays	20 - 200	SSD	6 - 48 T	128 - 768
Within Racks	200 - 1000	SSD	300 - 2400 T	2,000 - 16,000
Among Racks	1000 - 5000	SSD	1 - 100 P	32,000 - 1,600,000

Memory Maps 07/25/15

It is important to note that we are measuring time delays (nanoseconds), not bit transfer rates or frequencies (Gigabytes/second). Frequencies may remain constant over long distances. But in a computer, delays determine speed, and do not remain constant. For various reasons, they tend to increase exponentially with distance, mainly due to circuit designs that are required to maintain a recognizable signal. When going beyond a board, e.g., one with 32 processors, one is faced with delay multipliers on the order of 20 compared to level 1 cache.

Figure 16-1 provides a simplified illustration of the range of differences in memory boundary crossing delays when transferring smaller blocks of memory. To simplify the analysis, the memory access delay between a processor and its level 1 cache is set to 1. Going to level 2 and 3 cache increases the delay to 2 and 4. Once one is off the board, going between boards, then trays, and then racks, delays start to climb, depending upon distance as well as the hardware itself. The numbers in the chart are only intended to illustrate a hardware architectural delay footprint. They can grow with distance between racks. Clearly, one wants to remain as close to the diagonal as possible to achieve fast memory transfers. Except for embarrassingly parallel applications, information must be shared between boards, trays, and racks. The best software architecture minimizes off-diagonal memory sharing and transfers.

PROCESSOR	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
1	1	2	2	2	4	4	4	4	8	16	16	16	16	16	16	16	16	32	32	32	32	64	64	64	64
2	2	1	2	2	4	4	4	4	8	16	16	16	16	16	16	16	16	32	32	32	32	64	64	64	64
3	2	2	1	2	4	4	4	4	8	16	16	16	16	16	16	16	16	32	32	32	32	64	64	64	64
4	2	2	2	1	4	4	4	4	8	16	16	16	16	16	16	16	16	32	32	32	32	64	64	64	64
5	4	4	4	4	4	1	2	2	8	16	16	16	16	16	16	16	16	32	32	32	32	32	32	32	32
6	4	4	4	4	4	2	1	2	8	16	16	16	16	16	16	16	16	32	32	32	32	32	32	32	32
7	4	4	4	4	4	2	2	1	8	16	16	16	16	16	16	16	16	32	32	32	32	32	32	32	32
8	4	4	4	4	4	2	2	2	8	16	16	16	16	16	16	16	16	32	32	32	32	32	32	32	32
9	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	16	32	32	32	32	32	32	32
10	16	16	16	16	16	16	16	16	8	2	1	2	2	4	4	4	4	16	32	32	32	32	32	32	32
11	16	16	16	16	16	16	16	16	8	2	2	1	2	4	4	4	4	16	32	32	32	32	32	32	32
12	16	16	16	16	16	16	16	16	8	2	2	2	1	4	4	4	4	16	32	32	32	32	32	32	32
13	16	16	16	16	16	16	16	16	8	4	4	4	4	4	1	2	2	2	16	32	32	32	32	32	32
14	16	16	16	16	16	16	16	16	8	4	4	4	4	4	2	1	2	2	16	32	32	32	32	32	32
15	16	16	16	16	16	16	16	16	8	4	4	4	4	4	2	2	1	2	16	32	32	32	32	32	32
16	16	16	16	16	16	16	16	16	8	4	4	4	4	4	2	2	2	1	16	32	32	32	32	32	32
17	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	
18	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	1	2	2	2	4	4	4
19	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	2	1	2	2	4	4	4
20	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	2	2	1	2	4	4	4
21	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	4	4	4	4	1	2	2
22	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	4	4	4	4	2	1	2
23	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	4	4	4	4	2	2	1
24	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	4	4	4	4	2	2	2
25	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	8	8	8	8	8	8	8
26	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	16	16	16	16	16	16	16
27	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	16	16	16	16	16	16	16
28	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	16	16	16	16	16	16	16
29	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	16	16	16	16	16	16	16
30	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	16	16	16	16	16	16	16
31	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	16	16	16	16	16	16	16
32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	16	16	16	16	16	16	16
33	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
34	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
35	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
36	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64

CHIPBOARDTRAYRACK

Figure 16-1. A simplified illustration of memory boundary crossing delays.

From Figure 16-1, one sees that a good software architecture on a 32 processor PC board may exceed the speed of a computer using 10 or more times the number of processors. This depends upon the application and the software spaces designed to support the algorithms for the application. Clearly one must take careful measurements to make these comparisons.

MAPPING SOFTWARE ARCHITECTURES ONTO HARDWARE ARCHITECTURES

The architecture of a TELEPHONE_NETWORK Model is shown in Figure 16-2. This architecture contains 4 IND modules and corresponding IP resources (blue borders). Figure 16-3 provides a connectivity matrix of the architecture that has already been diagonalized.

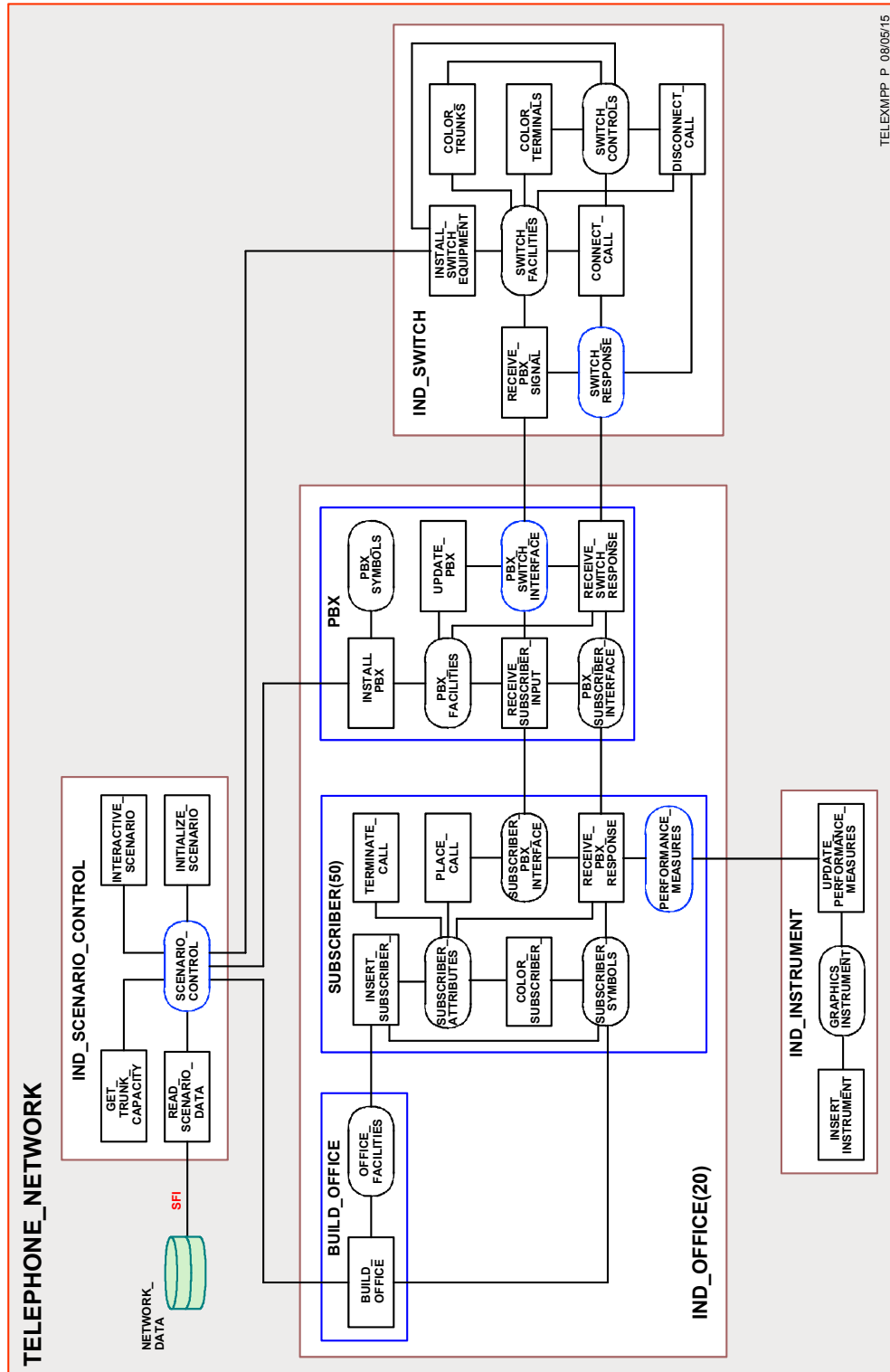


Figure 16-2. TELEPHONE_NETWORK Model.

In the matrix in Figure 16-3, SCENARIO_CONTROL, OFFICE, SWITCH, and INSTRUMENT, are IND modules and the IP resources shared between these modules are only read by processes outside the IND modules that write to them. The Xs indicate shared resources that are both read and written by processes within the IND modules shown. The Rs indicate IP resources that are only read by processes outside the IND modules.

IND MODULES															
MODULES	PROCESSES	RESOURCES													
		SCENARIO_CONTROL	OFFICE_FACILITIES	SUBSCRIBER_SYMSMBOLS	PERFORMANCE_MEASURES	SUBSCRIBER_ATTRIBUTES	SUBSCRIBER_PBX_INTERFACE	PBX_SUBSCRIBER_INTERFACE	PBX_FACILITIES	PBX_SYMSMBOLS	PBX_SWITCH_INTERFACE	SWITCH_RESPONSE	SWITCH_FACILITIES	SWITCH_SYMSMBOLS	GRAPHICS_INSTRUMENT
READ_SCENARIO_DATA		X													SCENARIO_CONTROL
INITIALIZE_SCENARIO		X													
INTERACTIVE_SCENARIO		X													
GET_TRUNK_CAPACITY		X													
BUILD_OFFICE		R	X	X											OFFICE
INSERT_SUBSCRIBER			X	X											
PLACE_CALL						X	X								
TERMINATE_CALL				X		X									
COLOR_SUBSCRIBER				X		X									
RECEIVE_PBX_RESPONSE				X	X	X	X	X							
RECEIVE_SUBSCRIBER_INPUT							X	X	X		X				
INSTALL_PBX		R							X	X					
UPDATE_PBX									X		X				
RECEIVE_SWITCH_RESPONSE								X	X		X	R			
RECEIVE_PBX_SIGNAL											R	X	X		SWITCH
INSTALL_SWITCH_EQUIPMENT		R											X	X	
COLOR_TRUNKS													X	X	
COLOR_TERMINALS													X	X	
CONNECT_CALL											X	X	X		
DISCONNECT_CALL											X	X	X		
UPDATE_PERFORMANCE_MEAS					R										X INSTRUMENT
INSERT_INSTRUMENT															X

CONNECTIVITY_MATRIX 11/29/13

Figure 16-3. Illustration of a diagonalized connectivity matrix.

Although this is a simple model, it illustrates the natural ability to create IND modules of physical systems. This example shows how an ASA is used to map the inherent parallelism in an application into a software space that fits into the type of hardware architecture required for parallel processing.

Past experiments have shown that adding more processors can cause substantial reductions in PUE, and corresponding reductions in speed multipliers that one may expect from a large increase in the number of processors used for an application. We will consider some of these applications using the ASA approach.

Fine Grain Model Software Architectures

Typical applications justifying large numbers of processors are the ‘fine grain’ problems of fluid dynamics or particle physics used to represent biological, chemical and meteorological type systems. Chapter 9 presented different approaches to software architecture of fine-grain problems. In these types of models, one is typically concerned with the dynamics of particles under the influence of one or more fields, e.g., gravitational, electro-magnetic, pressure and temperature. These systems are typically represented by a large number of cells in a 3D space that are used to discretize the system of partial differential equations that represent the smoothed dynamics of the system elements.

A simplified example of the interfaces between such cells is shown in Figure 16-4. When using 3D models, each cell has 6 interfaces, one in each of the positive and negative (X, Y, Z) directions. In the case of container or surface boundaries, the number of faces of cells adjacent to the container is reduced.

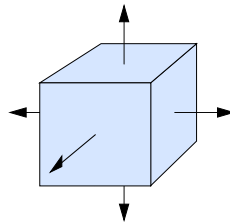


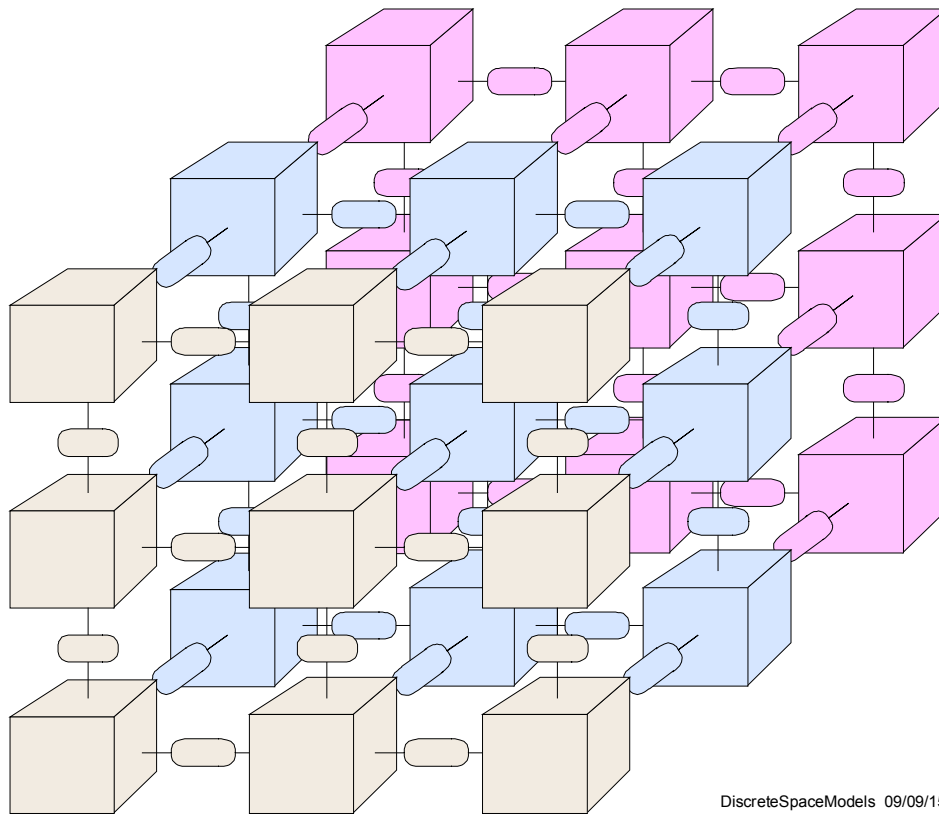
Figure 16-4. Illustration of a cell that is connected through 6 faces in (X, Y, Z) space.

The amount of computation within a cell will depend upon the particular dynamics within that cell. Computational imbalances among cells will reduce the PUE. Depending on their distribution, these imbalances may be offset by grouping a large number of cells onto a single processor. This grouping, often referred to as ‘tiling’, is described near the end of Chapter 9. Grouping multiple cells on a processor can also speed the overall process by increasing the amount of computation on each processor within each time interval while minimizing information exchanges at each processor interface.

Determining the best grouping is a software architectural issue that requires experimental evidence and comparison. This can be accomplished using previously validated models in a simulation, where the architectures are easily varied. In the end, one typically groups a large number of cells on each processor that interfaces with other processors to cut down on the memory boundary crossing delays.

A Simplified Hardware Architecture Model

To understand the translation of software architecture into a parallel processor hardware architecture, we will use a conceptual model of the hardware space as shown in Figure 16-5. To explain the important concepts behind the translation we start with a highly simplified hardware model using boxes to represent a single processor with information only shared directly between the 6 faces. This eliminates the hierarchy of chips, boards and trays and the need to account for complex hardware communication facilities that provide for “direct” communications (memory transfers) between any two boxes. The concepts are easily extended to more complex boxes.



CONCEPTUAL ARCHITECTURAL EXAMPLE:

A matrix of 27 major boxes, each containing a single processor.
 If each processor contains $54 \times 54 \times 54 = 157,464$ cells, one can model 4,251,528 cells.
 A single resource is shared between the adjacent face of each major box.

Figure 16-5. A 3D array of major boxes.

The boxes in Figure 16-5 represent hardware facilities where each box contains a single processor. If within each box the number of cells on that processor is equal to $N \times N \times N$, where N represents the number of cells along each edge, then Figure 16-6 provides a plot of the interior and face cells for various values of N . For example, if $N = 54$, yielding $54 \times 54 \times 54 = 157,464$ cells, and the ratio of interior cells to those on the faces is greater than 10:1. This ratio is important since interior cells share data directly whereas those on a face must share data between boxes. In this example, the total number of cells processed by the 27 boxes is 4,251,528

To analyze the memory boundary crossing delays one may encounter when dealing with a fine grain application, we will consider a simplified hardware architecture where memory is copied directly between the boxes via resources as shown in Figure 16-5. To share data between boxes, resources must be copied at each interface. We will assume that the time to copy a resource between boxes (processors in this case) is much larger than an internal memory access.

This architecture can be replaced by that shown in Figure 16-7, where the box in the center of Figure 16-5 has direct access to every other box. We emphasize that these are simple generalizations to illustrate critical hardware considerations.

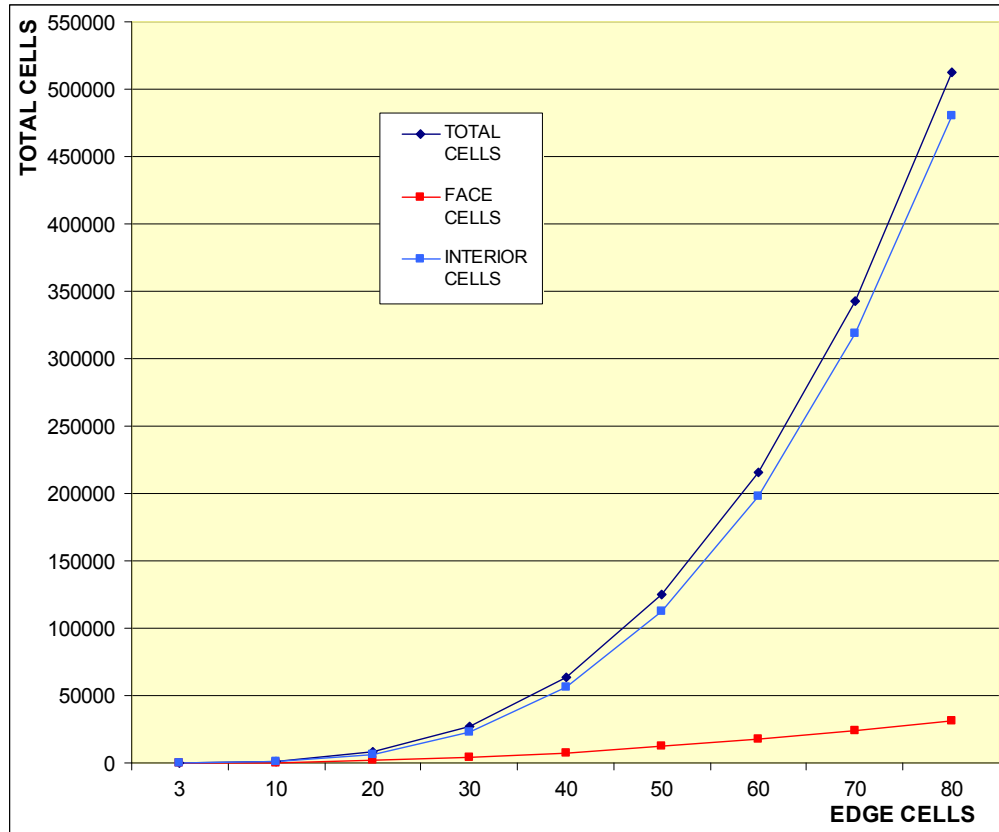


Figure 16-6. Total cells as a function of edge cells.

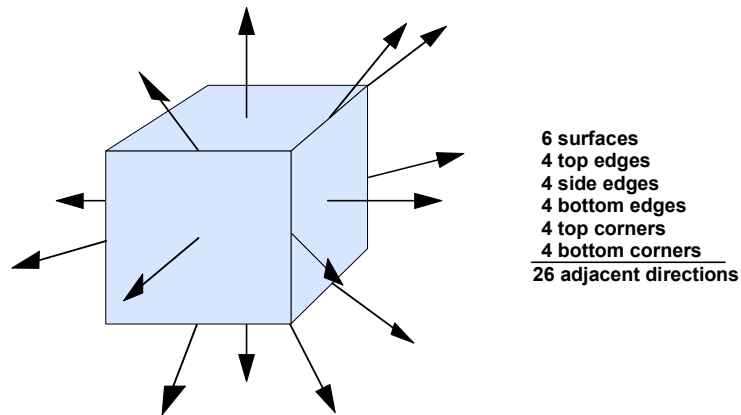


Figure 16-7. Passing information to surrounding boxes.

Modeling The Application

As indicated above, fine grain applications typically involve effects (e.g., gravitational forces) that affect cells that may be a distance of 10 cells away. The actual effects are determined by the size of a cell, the masses in a cell, and the amount of influence that surrounding cells have relative to those that are distant. These are typical modeling problems that require the knowledge of application experts. The models must be built to accommodate the accuracy requirements of the application - independent of how they are implemented on a computer. The accuracy is determined by the cell sizes, and how the effects of distance and other factors determine which cells affect the computation on the cell of interest.

In a typical force field, effects travel at the speed of light, implying that the effects of a change in mass in one cell will occur virtually instantaneously in cells that may be multiple hops away. This implies that, when masses move in a given cell, they can affect the forces in a cell that is multiple hops away. The mass changes occur within a ΔT time step, where ΔT is defined to be sufficiently small to accurately depict the physical changes in position. At the end of that ΔT , the new positions must be invoked “instantaneously” in all affected cells. Translating this to the solution of a set of partial differential equations, this implies that, as masses move within a time step, the new mass positions must be used in the next time step by those cells affected. This implies that a cell affected by the characteristics of cells 6 hops away must have access to those memory resources attached to the string of 6 cells. Obviously this applies in all directions.

Stepping back for a look into real memory boundary crossing delays, if all the cells affecting the computation in a given cell are on the same processor, they are likely sharing the same L2 cache memory. If all the cells are on different processors on the same chip, they are likely sharing the same L3 cache memory. In the case we are currently considering, if the cell of interest is on the face of a box, then it must communicate with cells in an adjacent box. Using current hardware designs, this memory boundary crossing delay is typically large compared to the others and must be dealt with accordingly. This is due to the design of communications channels between boxes over large physical areas.

Going back to the single processor in a box, the solution to instantly sharing information between boxes lies in creating an application space that supports fast memory transfers when mapped into the hardware space. Cells accessing data on cell faces that are 6 layers away, but within the same processor, can share one large resource (data structure) and its corresponding block of memory.

When cells must share data between processors (boxes), they must share information on cells up to 6 layers deep at the face of the block. This can be done using large data structures (resources) at the shared faces between boxes as shown in Figure 16-5. Given that this resource contains all of the information needed by those cells within 6 cells of the corresponding face on the adjacent processor, one need only copy this resource to the adjacent processor at the end of each time step. This IP resource must contain all of the information needed to account for the effects of those cells within the affecting facial distance on the adjacent processor.

The information put into this adjacent cell block can contain predetermined values that apply to those cells affected in the other box, reducing the computation required in the affected cell. Also, IP resources must be created to support cells on the four diagonal corners that are not directly adjacent, but must be passed to those four boxes by the adjacent box. Obviously, IP resources are required for each direction from which the forces emanate.

Hardware Memory Design Considerations

Two points can be derived from the above observations and each of these leads to the desire for large amounts of fast memory next to each processor. First, the number of cells along each edge must exceed the number of cells affected by a given cell in each direction by an amount that sufficiently reduces the overall memory transfer and computation time. This will also improve the ratio of internal cells to those cells affected by their nearness to a face.

If the number of edge cells in a processor is large, then the number of times that one must communicate between boxes is small relative to the overall computations. From the curve in Figure 16-6, if the edge contains 54 cells, then there are 10 interior cells for every edge cell. In addition, one must consider the number of cells in a given direction that contribute to the calculation of the cell of interest. This must be tested to determine the actual outcomes because of the complexity of the operations leading to the resulting effects. The major factor determining speed within the processor is the number of cache faults that occur when performing the calculations. The amount of fast cache directly available to a processor will determine the number of cells that can be processed fast on that processor. We cannot over emphasize the potential gains in speed achieved by increasing memory size as close to the processors as possible.

The second observation point is the time to transfer memory between boxes at a face. Given that the speed of applications of interest is enhanced when memory is only shared directly between the 6 faces of a box, then there is no need for a special communication interface that goes beyond the adjacent box. This implies that one may use direct memory transfers between boxes using IP resources - just as between processors on different chips on the same board. These transfer mechanisms need not be identical in speed, but the logic and circuit designs should produce transfer times that are as close as possible. The critical point is that memory transfers between adjacent boxes should be as fast as possible. When application architectures are designed correctly for speed, including the use of complex hierarchical IP resources, fast memory transfers between adjacent boxes is sufficient. Multiple hops can still be achieved with fast transfer times when the hop count is small. This satisfies most all applications of interest, particularly since the 2 to 4 orders of magnitude of speed gained satisfies these applications within an array of boxes smaller than that illustrated in Figure 16-5.

Given that IP resources can be used between boxes with memory transfer rates that are much faster than when using communication protocols, and that a sufficiently large set of applications exist that benefits from this approach, then this hardware architecture should be very attractive from an economic standpoint. This becomes more apparent when reflecting back to the case of multiple processors on a chip and multiple chips on a board. Today's boxes can hold 72 processors instead of 1. Most important is the fact that speed can be dramatically increased - by 2 to 4 order of magnitude - using the VisiSoft CAD approach. This implies huge reductions in the number of processors required to service the vast majority of applications.

One must also consider the requirement for all boxes to transfer memory to all other boxes. This brings in the need for communication protocols that move memory from anywhere to everywhere. It is rare that simulations of physical systems have this requirement, including models of communication systems. This is also true for real-time as well as other applications. Administrative functions that service the computer system will likely have such requirements, but these may be supported using the same approach, putting the protocols into software and treating these administrative functions as another application.

Alternatively, separate communication hardware may be accommodated within each box, with messages and data routed between the boxes. These administrative needs will likely not have the typical speed requirements of a large scale simulation. Even if they do, the number of boxes required in a typical installation serving current application requirements will be significantly reduced compared to those using current software approaches.

Effects on Parallel Processor Utilization Efficiencies (PUEs)

The effects of this approach on Processor Utilization Efficiencies (PUEs) is illustrated in the example shown in Table 16-2 and Figure 16-3 below, where 1,000,000 fine grain cells may be grouped onto much fewer processors. Starting with a single cell per processor and moving up the logarithmic scale to 1,000,000 cells per processor, the table illustrates the potential for order-of-magnitude increases in speed using VisiSoft. With large numbers of processors, typical PUEs using current approaches are on the order of 7% - 10%. But the Speed Multipliers (SMs) are based upon single processor speeds that may be 10 to 100 times slower than VisiSoft for the same application - on the same processor. Thus the effective SM and PUE must be calculated based upon the VisiSoft single processor speed. NOTE: This will drastically change the speed multipliers calculated by other organizations.

Table 16-2. Grouping fine grain cells onto fewer processors.

CELLS PER PROCESSOR	Number Of Processors	Typical PUE	Typical Speed Multiplier †	VisiSoft PUE	VisiSoft Speed Multiplier †
1	1,000,000	0.05	50,000.0	0.50	5,000,000
8	125,000	0.06	7,500.0	0.60	750,000
64	15,625	0.07	1,093.8	0.70	109,375
216	4,630	0.08	365.7	0.79	36,574
512	1,953	0.09	166.0	0.85	16,602
1,000	1,000	0.09	89.0	0.89	8,900
8,000	125	0.09	11.6	0.93	1,163
64,000	16	0.10	1.5	0.96	150
216,000	5	0.10	0.5	0.98	45
512,000	2	0.10	0.2	0.99	19
1,000,000	1	0.10	0.1	1.00	10

† Adjusted to use the fastest single processor speed.

With 1,000,000 processors, the VisiSoft PUE may be reduced to 50% due to the footprint of the machine. But because VisiSoft single processor speeds are typically 2 orders of magnitude times faster than other languages, one can put 100 times the number of cells on a given processor. Putting 216,000 cells (60X60X60) on a processor is easy using VisiSoft data structures. One then achieves a speed multiplier of 45 using only 5 processors. One would likely have to use more than 100 times that number of processors to achieve that speed using typical approaches. As cells per processor increases, the requirement for more memory closer to the processor rises.

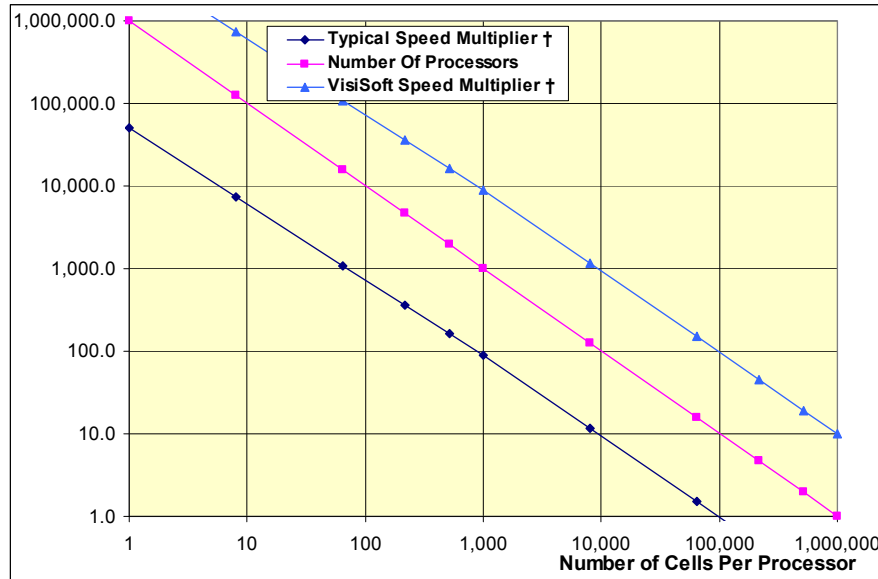


Figure 16-8. Grouping fine grain cells onto fewer processors.

As illustrated in the table, the PUEs will rise as the footprint gets smaller. Again, the smaller multiplier of 700 may likely be acceptable since the size of the computer is reduced by a factor of 27, substantially reducing the cost of equipment, maintenance, floor space, environmental equipment sizes, and power consumption. By grouping 216,000 cells into a processor, both the speed multiplier and the number of processors is reduced by almost a factor of 3. One can easily make the trade-off of speed versus cost in a reasonable range of the cell-processor spaces. By using less processors, the information exchange overhead between processors is obviously significantly reduced.

Using this approach, if the system is linear, each block of cells is doing a significant amount of work independent of the other blocks of cells. If the system is nonlinear, then one can estimate the values for the nonlinear interfaces as a starting point and iterate if necessary using a fast converging linear segmentation algorithm that may be used for all of the affected cells within each ΔT . In either case, the amount of computation within each processor will be large compared to the cross processing necessary to update and synchronize the interface processing.

Finally, one must be able to easily map blocks of cells onto the hardware architecture, to minimize the memory boundary crossing delays described in Table 16-1 and Figure 16-1. This is a software architectural problem that must be solved to minimize the run times. We note that problems such as this are easily simulated to obtain representative test data. To obtain valid results, one must match the model parameters to live test data using parametric analysis, a standard technique for producing accurate models and simulation results.

Mapping Software Spaces Into Hardware Spaces

Mapping cell blocks in software (x, y, z) space into hardware (x', y', z') space is relatively easy when the application space is a rectangular tank. So let's consider mapping other spaces, e.g., (R, Θ, Φ) - or even sets of different connected spaces - into hardware (x', y', z') space. We start with a set of different connected spaces shown previously in the waveguide application in Figure 9-17 (copied below as Figure 16-9). The wave guide presents an application space architecture that uses multiple complex connected spaces.

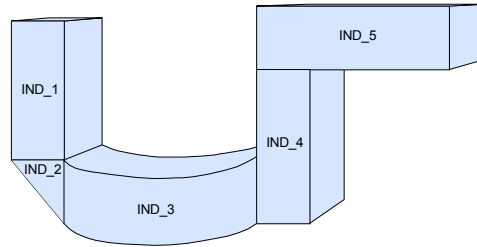


Figure 16-9. Application Space Architecture

Given that the application is running on a single processor, the spatial mapping has already been completed, including design of the resources that connect the sections. Thus the major effort must be on breaking these sections up to run on separate processors. Although this looks difficult, it is easily mapped into an array of boxes as shown in Figure 16-10. Given that the waves travel in a given manner within each space, with reflections occurring in each space, the hardware space simply follows the application space. Each box represents one - or a part of one - of the application spaces.

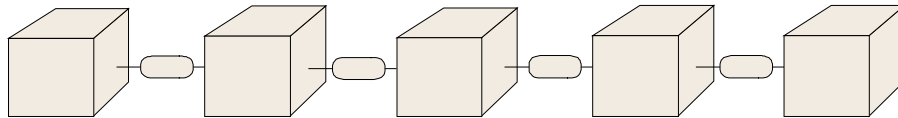


Figure 16-10. Hardware Space Architecture.

Global Planning

The space that dictates the operations of the next set of applications is a representation of the earth itself, see Figure 16-11. Although not immediately apparent, Figure 16-5 provides the hardware space in which to map these global applications. The earth's surface maps close to a sphere with (R, Θ, Φ) coordinates. For applications requiring more accuracy, such as those described below, Θ is the Latitude coordinate running from $(-90^\circ$ to $+90^\circ)$ corresponding to boxes running from bottom to top in the drawing. Φ is the Longitude coordinate running from $(-180^\circ$ to $+180^\circ)$ corresponding to boxes running from left to right and around the back. In addition, each box has an Altitude coordinate that is perpendicular to the surface of the earth. All points above the earth's surface are described in terms of (LAT, LON, ALT).

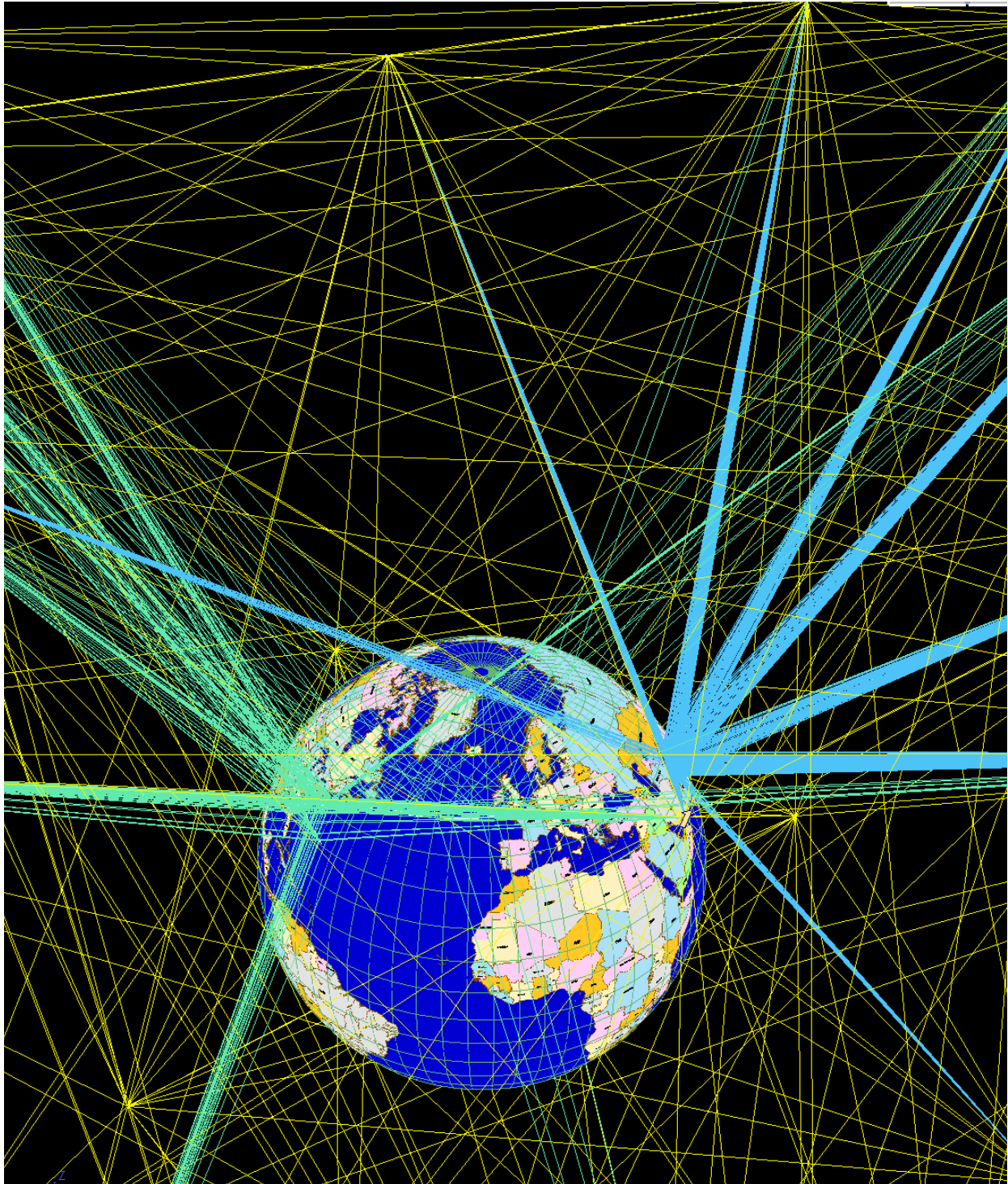


Figure 16-11. Connectivity between satellites, ships, aircraft, and ground vehicles.

Figure 16-11 illustrates a complex communications space around the globe. To speed up the models on a single processor, the earth's (LAT, LON, ALT) coordinates are transformed into a large number of sets of (x, y, z) coordinates mapped over the earth's surface. This eliminates sine and cosine calculations since waves travel through space in straight lines.

Figure 16-11 illustrates satellites in the GPS constellation. Depending upon the level of model detail, or the need for other spatial platforms, e.g., communication satellites, an additional layer of processors may be needed to represent platforms in the space layer. This is supported by adding boxes around the center of the periphery as shown in Figure 16-12, yielding a total of 32 boxes.

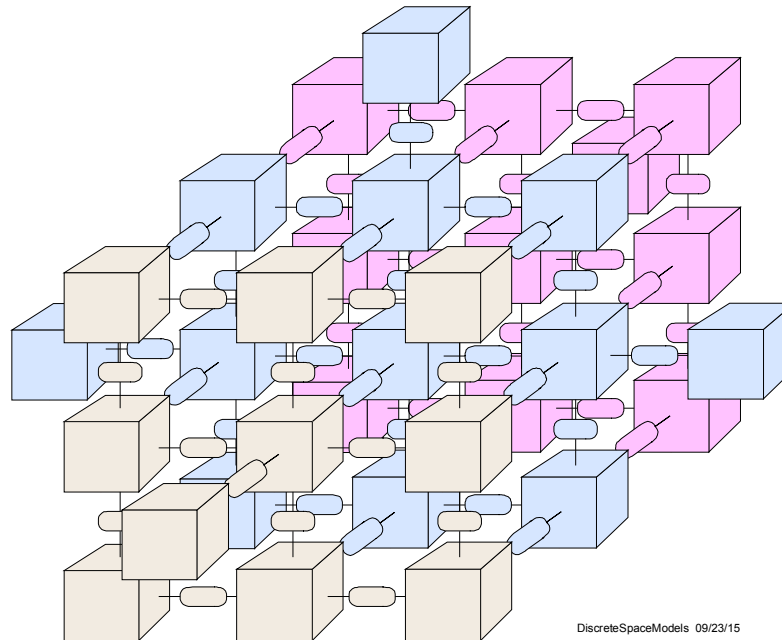


Figure 16-12. Representing the earth's surface and the space above it.

Figure 16-13 shows another application based on the earth's surface and its surrounding layers. It is the computation of signal power received on the surface of the earth from an antenna on or near the surface of the earth using transmitters in the HF frequency range. These signals bounce off the ionosphere and may be received far from transmitters on the other side of the earth. These receptions depend upon the position of the earth relative to the ionosphere which are functions of time and date. These calculations use application spaces similar to those described above. Consequently, they map conveniently into the hardware space similar to that shown in figure 16-5 or 16-12.

Whether working in terms of (R, Θ, Φ) coordinates, (LAT, LON, ALT) coordinates or sets of (x, y, z) coordinates, the mapping of all these application spaces into a hardware space similar to that shown in Figure 16-5 is easily accomplished. More importantly, the model spaces are all designed for maximum speed of the transformations as well as the application algorithms.

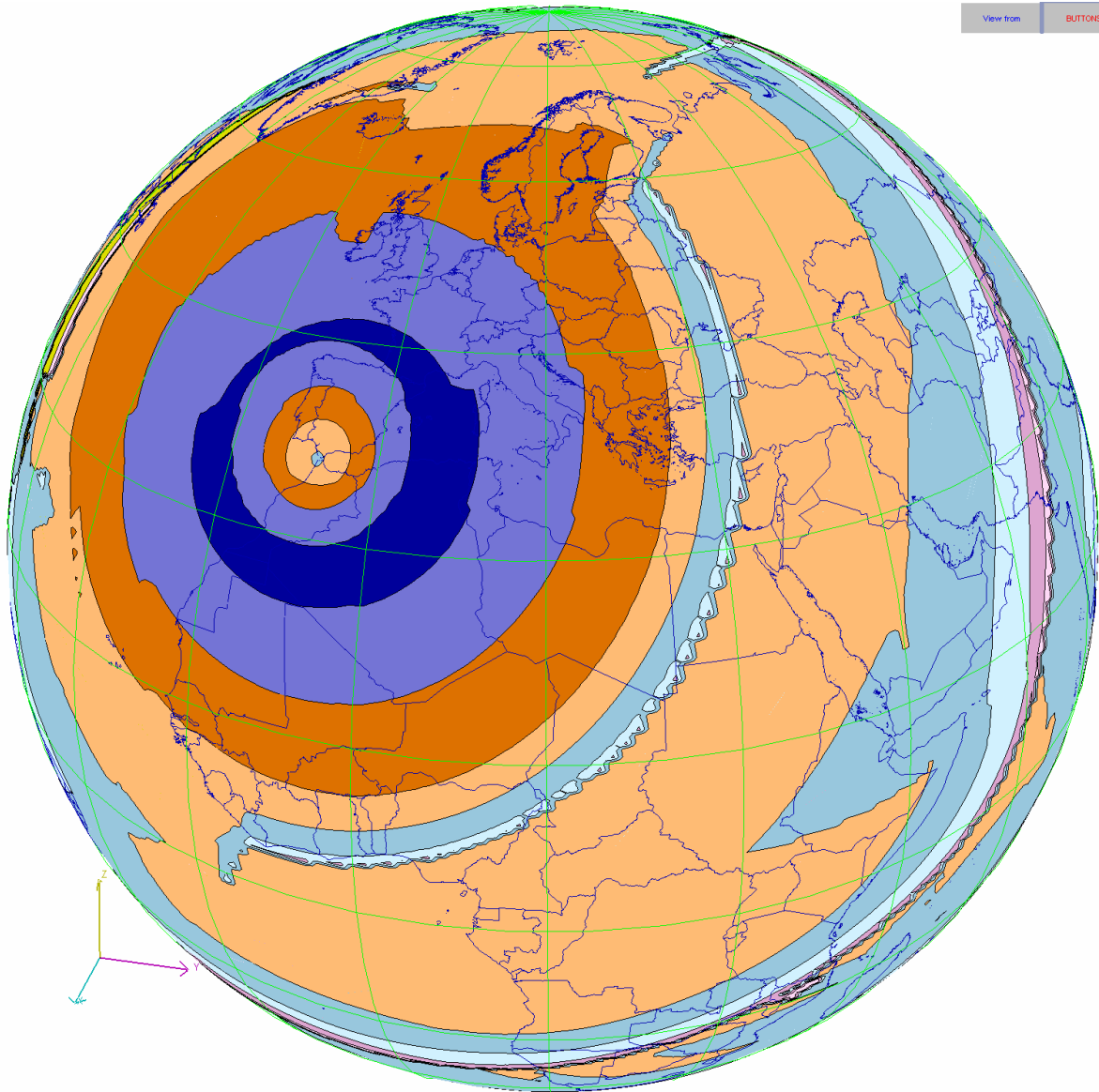


Figure 16-13. HF signal strength around the globe from antenna in Morroco.

The Bottom Line

The combination of software and hardware architectures for the above applications can provide speeds that are likely impossible to achieve using any other technology today, forgetting the corresponding power consumption and cost of operations. The discounted cost of this full up global analysis and planning facility will be less than \$5M. This implies it will likely be many times faster - at less than $1/100^{\text{th}}$ of the operational costs and $1/20^{\text{th}}$ of the initial cost - of any other approaches.

VSI's wholly owned subsidiary, the Green Gene Machine Corp., is currently working with universities to prove these claims independently.

QUESTIONABLE HARDWARE APPROACHES

There are a number of questionable hardware design approaches that have come about to make up for the poor processor utilization efficiencies described in Chapters 6 and 8. These approaches have been devised to make up for the poor software architectures that are typically used for parallel processors. One of these techniques is hardware cache coherency. With a simple architectural approach, VisiSoft memory consistency and synchronization is guaranteed while speed is enhanced. Another questionable approach is providing memory transfers from all processors to all processors. This invokes complex communications protocols implemented with special hardware. This requirement is unnecessary for most - if not all - parallel processor applications known to the authors. Both cache coherency and all to all communications are totally unnecessary using VisiSoft. This precious chip space can be used for more memory close to the processors - increasing speed and reducing the footprint.

Other approaches look to build larger systems with increased numbers of processors, as if running time increases directly with more processors. These systems use Gigabit LANs and switches that create communication delays. The delays are caused by:

- High Overhead
- Huge Memory Boundary Crossing Delays
- Nonlinear distance multipliers

These approaches work well if there is little or no communications between processors, essentially *the embarrassingly parallel case*. Given a good application space architecture, all that communications software is virtually unused, except for administrative functions. But in some organizations, the current race appears to be focused on how many processors one can string together in a single computer. When one reads about the test runs on these machines, fair measures of speed multipliers and processor utilization efficiency are hard to find, see [8]. Yet these are the critical economic measures for comparing parallel processors.

The Goal Of Hardware Approaches

The goal of hardware designers should be to optimize the economic case for the end users, i.e., minimizing the cost to map the inherent parallelism of applications onto a computer. This is depicted in Figure 8-3. This chain starts with the development environment to create a software architecture that maps the inherent parallelism into IND modules, and provides the information required by the Run-Time System (RTS) to map those modules into an optimal hardware configuration. As described above, memory is the key factor in obtaining speed for most types of problems. This includes minimizing memory boundary crossing delays and overhead that can be exchanged for more memory close to the processors.

There are three significant factors affecting parallel processors speeds. The first is using a fair basis to compare speed multipliers (e.g., using the fastest single processor speed). The second is mapping the inherent parallelism of an application into software spaces that simplify the algorithms so they run fast. Third is the Processor Utilization Efficiency provided by the fit of software and hardware architectures. The consequences of PUE are illustrated in Figure 16-14. The product of these and other measurable effects determines the multiplier on speed of an application running on a parallel processor.

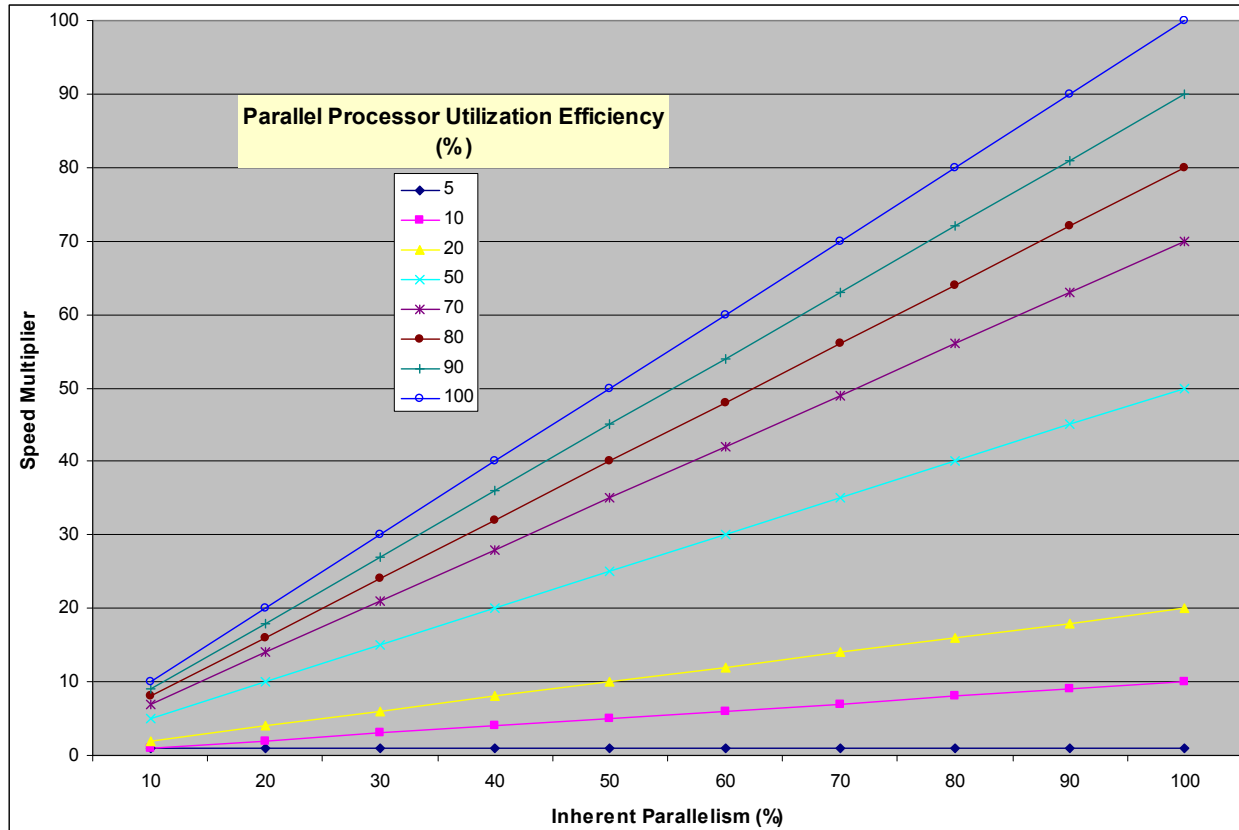


Figure 16-14. The effect of parallel Processor Utilization Efficiency (PUE).

Figure 16-14 is an extension of Figure 8-3 based upon additional information. Figure 8-3 assumes that both the hardware and software design achieve a perfect translation of the inherent parallelism in the system. This is virtually impossible except for embarrassingly parallel applications. Figure 16-4 accounts for PUE which is treated as an additional factor affecting run-time speed, one that is influenced by both the hardware and software design. Given that the software design obtains a given translation of inherent parallelism for the RTS, one must be able to translate this via VPOS and the hardware into a final speed multiplier. Here we are looking at the PUE as a second factor that determines the final speed multiplier. This chart shows the effects of PUE given a software design that produces a given inherent parallelism. None of these efficiencies is going to approach perfection. But if the software design can produce PUEs greater than 90%, we are looking at multipliers of 8 to 9 on speed with a good mapping of the hardware onto the software.

With today's approaches, one considers "good" multipliers to be on the order of 7% to 10%. Granted, most of the poor results have been caused by the software design. But if a software design is produced that maps the inherent parallelism of an application into a software architecture that is highly effective, the hardware architecture will play a significant role in taking advantage of that software architecture to achieve substantially improved speed multipliers.

OTHER HARDWARE DESIGN CONSIDERATIONS

As described in Chapter 2, one of the most important considerations in computer design is the separation of servers from parallel processors. Server tasks can be divided into those that can be put on a parallel processor and those that are best running as a task on a single processor. A major factor in this division is the use of I/O devices and the design of DMA channels. Figure 3-6 illustrates a scenario where three parallel processing tasks are running concurrently on different sets of processors. These tasks may be managed by different servers that interface with different I/O devices.

A typical server application requires a teleprocessing front-end accessing a large database on the back end. The communications handlers and database handlers are best suited to the server environment. However, with the abundance of memory available on a parallel processor, one can maintain the total database in fast memory. Then most of the processing can be done on the parallel processor side. The database may be split into segments based upon statistics that provide a uniform access distribution across segments. Then separate processors may be allocated to each segment, providing fast turn-around times.

By creating independent modules during the architectural design, where threads are always independent between modules, and are sequential and cannot run concurrently within a module, thread synchronization is unnecessary. Using hardware (chip space) and OS code (memory) for this function is unnecessary for typical parallel processor applications.

Similarly, since synchronization between resources on separate processors is handled automatically by the RTS and VPOS, there is no need for hardware cache coherency. Using hardware (chip space) and OS code (memory) for this function reduces speed multipliers.

It is not clear how special hardware stacks save time. Since recursion is not known to be used in private sector applications, and certainly not needed in any software or simulation system known to the authors over the last 55 years, it appears unnecessary for the applications of interest here. It slows down computation while using precious chip space. This investigation must include the tradeoff of microcode versus direct implementation using logical circuitry.

From the above, one sees that software development environments affect the design of both the run-time environment and the operating system. Together, they both affect the design of the hardware. Given the VisiSoft CAD facilities described here, the following hardware facilities can be eliminated from parallel processor chips using the approach proposed here.

- DMA channel - I/O device interfaces
- Cache coherency
- Thread synchronization
- Stack facilities
- Special instruction swapping facilities

We note that these facilities can be replaced by more memory close to the processors, the major factor in achieving speed. We also note that the number of parallel processors in Figure 3-6 may be small compared to some actual environments. However, that configuration easily fits in a box the size of a PC.

SUMMARY

Translating an application with reasonable inherent parallelism into an effective parallel processing design is hardly different from solving complex problems in mathematical physics. It is much akin to development of the Open-GL graphical pipeline. It involves selecting and optimizing the best spaces to do the transformations, and organizing the transformations to gain speed. The application area expert is best equipped to solve this problem provided, as in similar fields, the expert is supported with a CAD system. The requirements for this CAD system start with the language that is used to describe complex spaces and transformations. It must provide the ability to create the detailed architectures required to understand and solve these types of problems. The Run-Time System must be optimized to take the architecture and work with VPOS to provide optimal mappings of the spaces and transformations into hardware architectures.

Given that these facilities can be used to provide highly effective solutions, they must reside on hardware that is designed to support them effectively. This is best accomplished by reviewing the applications destined to use the hardware, and designing hardware architectures that support the spatial mapping of the application

Clearly parallel processing stresses the language requirements well beyond a single processor application. Fortunately it forces the requirement for measuring the resulting designs by comparing speed. This brings about the obvious need for a scientific approach to the design of the CAD system as well as the OS and the hardware. All of these designs can be tested using simulation. This type of testing is addressed in the next chapter.

The bottom line here is that I/O bound applications require the special design of processors tailored to the server environment. In this environment, huge numbers of independent tasks are running concurrently on the server, many of which are running on the same processor. Since they are typically bound by substantial I/O operations, most of them are in a wait state while one is running. In other words, multiple independent *tasks* can run on a single processor since most are waiting for information transfers from other devices.

Parallel processor applications require speed independent of I/O operations. This does not imply that I/O is not used. It implies that a huge number of modules must share information directly while they run concurrently. Their I/O operations are infrequent, and can be supported by simplex channels (typically one-way output) that do not slow them down. Simplex channels can also provide one-way input, where the processors handling these channels provide the data - as needed - to modules that grab that data on the fly. This is where application area experts are required to design such architectures since only they know the constraints on the design and the corresponding approaches that can ensure maximum concurrency of operations.

To support these experts, they must be provided with a CAD environment to describe the facilities they need to ensure representing physical systems accurately. When systems are modeled along physical lines, they are easy for application experts to understand and are known to naturally operate as fast as possible.

CHAPTER 17.

SINGLE PROCESSOR TESTS & RESULTS

OVERVIEW OF EXPERIMENTS

Thirty years of productivity experiments, surveys, and speed testing would take many books to describe. In addition, surveys of people's ideas about productivity can easily be biased, with the real biases being almost impossible to remove using pure text book approaches. After one has heard the thoughts and witnessed the approaches of many people, the biases become large and obvious. It also becomes clear that the distribution of biased interpretations of test results can have variances as wide as the test results themselves.

This can be seen from the view of Jerry Sitner, a highly experienced software manager who has compared the productivity of different software development approaches over a huge number of real projects over many years. In his article "How much longer?" see [136], he compares his interpretation of IEEE standards for measuring productivity to his real world experience. Yet a PhD candidate in computer science with virtually no real project experience may have a totally different interpretation of a productivity experiment based upon student preferences than Sitner. When one is concerned with deriving test results that are on a sound economic and scientific basis, Sitner makes it clear that experiments can produce obvious results on real world software problems, particularly those that work with large databases.

Chapters 17 & 18 provide a few samples of the kinds of experiments and testing that have been performed by the authors. The results are hard numbers of clearly defined speed measures based upon the clock. The first experiment compares different approaches to writing code on a single processor, and the huge corresponding run time differences measured by the computer real-time clock. A by-product of this experiment is a comparison of the effort required to write fast code in C-based languages versus VisiSoft. We submit that it is hard to ignore the obvious differences in productivity (the coding differences get greater as the speed gets faster).

The second experiment, Chapter 18, compares results run on a parallel processor. Here we use the Windows "oscilloscope" backed up by times off the real-time clock to interpret the results. The purpose of this experiment is to compare measured data to the theoretical expectations of the RTS and VPOS designs. Again, the comparisons are obvious.

The purpose of these chapters is twofold. First is to demonstrate how experiments producing hard data can be used to put software theories on a sound experimental basis. It is the firm opinion of the authors that, by using such methods, software technology can be put on the same scientific footing as physics. Only then can the comments from engineers quoted in Chapter 1 be put to rest.

The second purpose is to show how simulation can be used to perform such experiments, just as is done in many engineering fields, e.g., electronic circuit design. With the CAD system described here, one can run simulated loading on a real parallel processor as well as simulate a parallel processor environment on a single processor. Anyone who has used CAD systems and simulations for design is well aware of the time savings - as well as many other benefits - of this approach, such as crafting models to sort out results from otherwise impossible experiments.

A SET OF TESTS FOR A SINGLE PROCESSOR EXPERIMENT

The tests for this experiment are designed to use a large database to illustrate the huge time differences that may be derived from different software languages. These tests show that the major factor affecting the simplification of transformations - and thus run-time speed - is design of the data space, a basic engineering principle. These tests prove that the organization of the data space into deep hierarchies gains the speed. As a by-product, it makes the design of the algorithms much easier to understand by a third party who may have to modify or reuse the code.

The single processor experiment is composed of a set of tests to read, transform, and write a large file containing triangles. The triangles are used to depict mountainous terrain in a graphical background overlay for the interactive military planning application described in Chapter 2. The software described below is taken from an advanced 3D graphical mapping system required for many new and future applications. This experiment is easily reproduced in a university environment, so that graduate, post graduate, or science and engineering staff may use different approaches to building software for specified applications and compare the run times. This experiment uses five simple tests that require no understanding of the application.

This experiment provides an introduction to the problem of testing the speed of software applications and critical software design factors that can cause significant changes in application run times. The experiment provides for reading and writing files of different sizes. This allows one to tune the experiment to support the two primary functions: (1) Testing to ensure that the experiment is being performed properly; (2) Testing to make comparisons of running times.

In general, an experimenter may read and write the files in any manner desired, so that different approaches may be compared in terms of speed. As a side benefit, one can get a feel for changes in productivity using different approaches. This is particularly true when comparing different software languages used to support the tests.

A Quick Look At The Underlying Application

It must be emphasized that an understanding of the application is not necessary to perform these experiments and make the speed comparisons. This section is provided only to satisfy the curiosity of the experimenter. It also provides an indication of the level of complexity of the application. The 3D terrain shown in Figure 17-1 is generated using the set of independent software tasks shown in the block diagram in Figure 17-2. It is important to note that this 3D terrain facility is but one of many background overlays in a complex military simulation involving hundreds of moving platforms.

To get a feel for the size of the mountains in Afghanistan and northwest Pakistan, the screen shot in Figure 17-1 covers an area about the size of Vermont and New Hampshire combined. This area is close to the Himalayas and contains peaks above 28,000 feet.

The speed comparison test uses a modified version of one of task T4 in Figure 17-2. Task T4 is used to generate inner-products between the triangle normals and a light vector. The speed test uses a very simple implementation that does not require an understanding of the application. This experiment principally tests different ways to read and write the files.

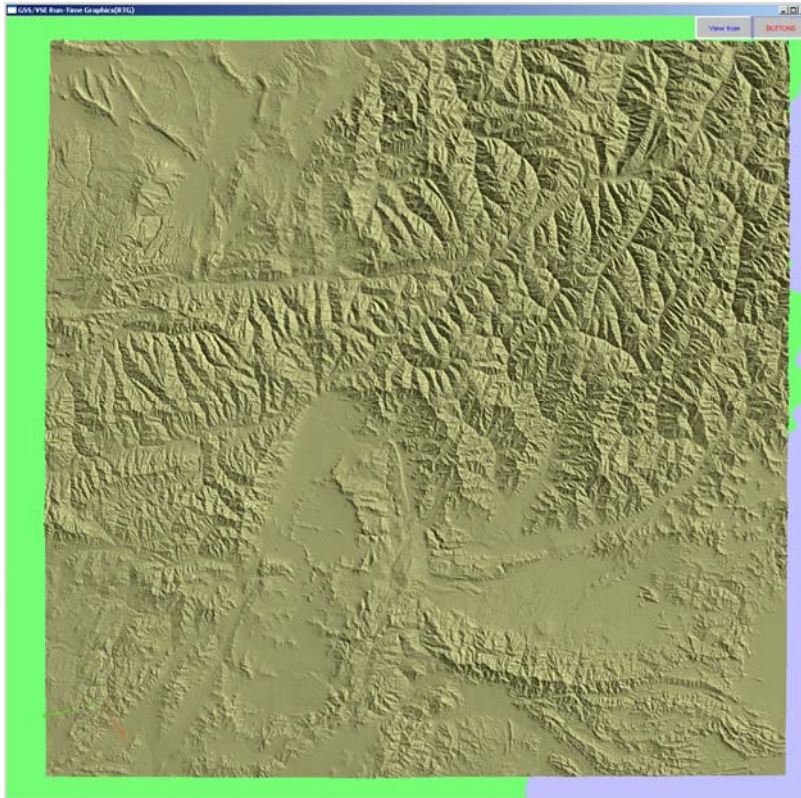


Figure 17-1. 3D terrain in Afghanistan.

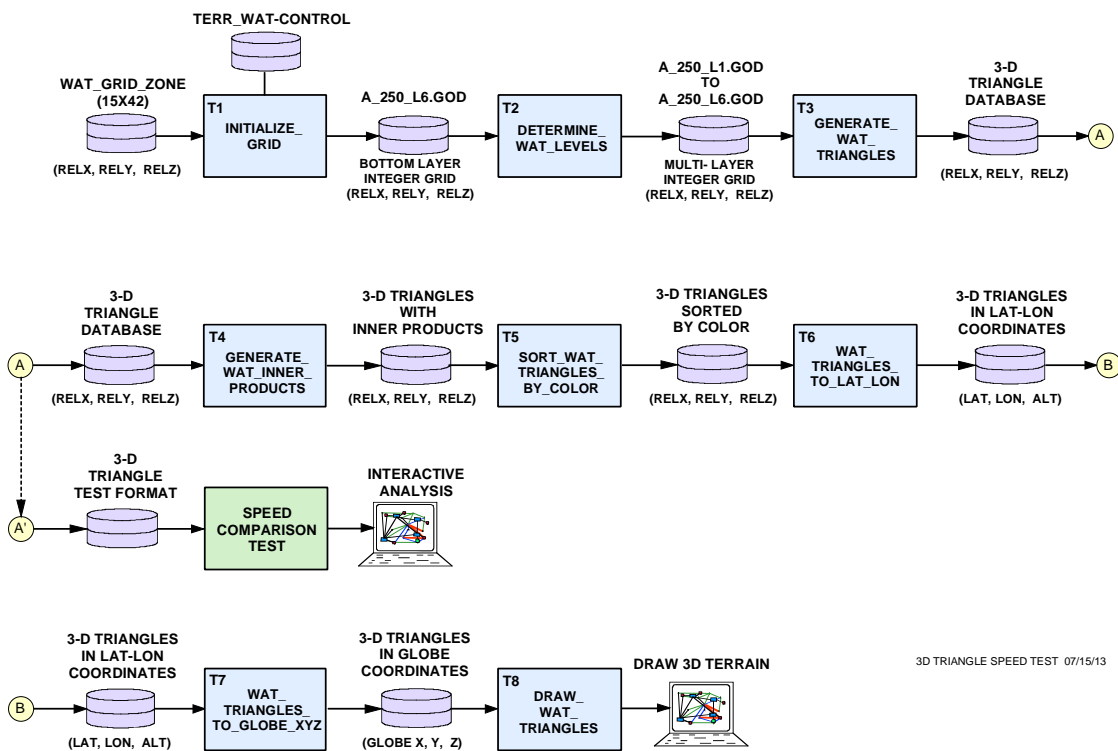


Figure 17-2. Block diagram of the 3D terrain production facility.

Overview Of The File Read-Write Experiment

Figure 17-3 is a block diagram of the software addressed in the experiment. The purpose is to create software for T3 and T4 that improves the speed of task, T4. T3 reads a file of triangle vertices and produces the INPUT TEST FILE for T4. T4 reads the INPUT TEST FILE and generates the output file of 3D triangle vertices along with inner products. The size of the input file may be varied to adjust the time to run the test. For example, the file required to generate the terrain shown in Figure 1 is about 500 Megabytes. This may be reduced by one or two orders of magnitude so that tests may be completed in a few minutes or seconds.

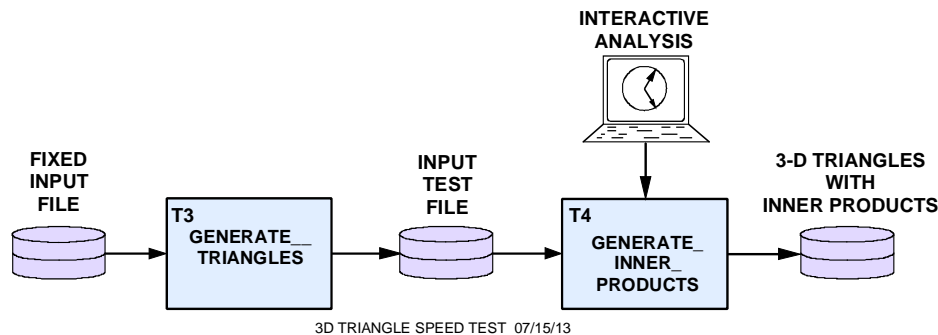


Figure 17-3. Block diagram of the 3D experiment.

The Fixed Input File contains two record types, G and T. The layout of these records is provided in Figures 17-4a & 4b. The experimenter must read this file in T3 and produce an Input Test File to Task T4. The experiment requires designing the software and files in and out of T4 to minimize the time it takes T4 to read and write the files. The T4 output file must contain the data in the two input records in Figure 17-4 while adding an INNER_PRODUCT field that is accurate to 3 decimal places, see for example INNER_PRODUCT_O in Figure 17-7(c).

FIELD CONTENTS	FIELD (Type Length)	FIELD DESCRIPTION
RECORD_TYPE	CHARACTER 1	Type Of Record: G
TRIANGLE_NUMBER	DECIMAL INTEGER 10	Triangle Number
LEVEL_NUMBER	DECIMAL INTEGER 1	Level Number
RIGHT_HAND_COLOR	DECIMAL INTEGER 2	0 = No color
RIGHT_HAND_REFLECTION	DECIMAL INTEGER 1	0 = No color
LEFT_HAND_COLOR	DECIMAL INTEGER 2	0 = No color
LEFT_HAND_REFLECTION	DECIMAL INTEGER 1	0 = No color
PAD	CHARACTER 21	Spaces to match record size

Figure 17-4a. Structure of TRIANGLE INPUT RECORD 1

FIELD CONTENTS	FIELD (Type Length)	FIELD DESCRIPTION
RECORD_TYPE	CHARACTER 1	Type Of Record: T
TRIANGLE_NUMBER	DECIMAL INTEGER 10	Triangle Number
VERTEX_NUMBER	DECIMAL INTEGER 1	Vertex Number (1, 2, or 3)
X_COORDINATE	DECIMAL INTEGER 9	Triangle Vertex Coordinate
Y_COORDINATE	DECIMAL INTEGER 9	Triangle Vertex Coordinate
Z_COORDINATE	DECIMAL INTEGER 9	Triangle Vertex Coordinate

Figure 17-4b. Structure of TRIANGLE INPUT RECORD 2

A printout of a sample of these records for two triangles is shown in Figure 17-5.

```
G00000000010000000
T00000000011000000000-00001000000016400
T00000000012000000250-00001000000016404
T00000000013000000125-00000875000016434
G00000000020000000
T00000000021000000125-00000875000016434
T00000000022000000250-00001000000016404
T00000000023000000250-00000750000016480
```

Figure 17-5. Sample records for two triangles

Information For Experimenters

All that is needed for this experiment is the Fixed Input File for Task T3 described in Figure 17-4. This file is provided as an ASCII (text) file as described in the record layouts in Figure 17-4 and sample in Figure 17-5. The size of the file is slightly more than 1 Gigabyte, but can be reduced for convenience of initial testing by the experimenter.

The accuracy of the terrain databases used to generate the triangle database is 100 meters. To be sure that the calculations do not reduce this accuracy, the numbers should all be represented with an accuracy of 10 meters. Since the Earth's radius is 6.378×10^6 meters, the numbers should be represented in fields larger than 638,000 meters. REAL numbers are represented accurately just beyond 2×10^6 , being well within the 10 meter accuracy requirement.

Given this file, the experimenter builds Task T3 to generate a file whose output format is up to the experimenter, and should be designed to help maximize the speed of Task T4. Task T4 must then be built to generate the 3D Triangle output file. The format for this 3D Triangle output file is up to the experimenter, but must contain all of the data from the fixed input file and the value of the inner-product.

It is necessary that the input test file be read into an intermediary structure where all of the triangle coordinates are REAL, since they are needed to do the inner product calculations. However, the experimenter should not compute the inner product for this experiment, but simply put the value 0.75 into that field in the output file.

It is suggested that the experimenter prompt for the number of triangles to be read from the input test file and terminate the task after that number has been written to the file. The prompting can begin before the time measurement starts.

The experimenter should measure the time it takes for each run, analyze the various times as different approaches are used, and draw conclusions about speed as a result of the changes.

Note: In the actual system, the inner product is calculated by determining the normal vector to the triangle, and then computing the inner product of that vector with the *reflection* vector (the negative of the light vector). This inner product is used to determine the color of the triangle's surface, yielding the 3D appearance shown in Figure 17-1. This calculation is not part of this experiment.

TEST DESCRIPTIONS & RESULTS

The test description provided above does not specify approaches to read and write the file. That has been left to the experimenter. Below we offer five different approaches, describe the theoretical reasons for our expected differences in outcomes, and present the test results.

Noting the size of the triangle database used for each of these tests (5,000,000 triangles), the approach to reading and writing the files will determine the speed. Our experience with large databases includes comparisons to competition in the time it takes to read huge files. Many previous comparisons have resulted in differences of 1 to 2 orders of magnitude (and times measured in seconds as opposed to minutes or hours). There are a number of factors that affect this. Each of these is described in the following tests.

Test 1 Description

The architecture for all of the tests are the same as that of TEST_1, shown in Figure 17-6. Test 1 uses the Fixed Input File/Record formats directly as specified in Figures 17-4 & 5. The output file is essentially the same as the input file with the exception that it contains the inner-product. To describe the data structures in detail, we use the VisiSoft resource formats precisely as they were used in the tests. Although these could be written in a C-based language format, they quickly become difficult to understand as the hierarchies in ensuing tests become more complex.

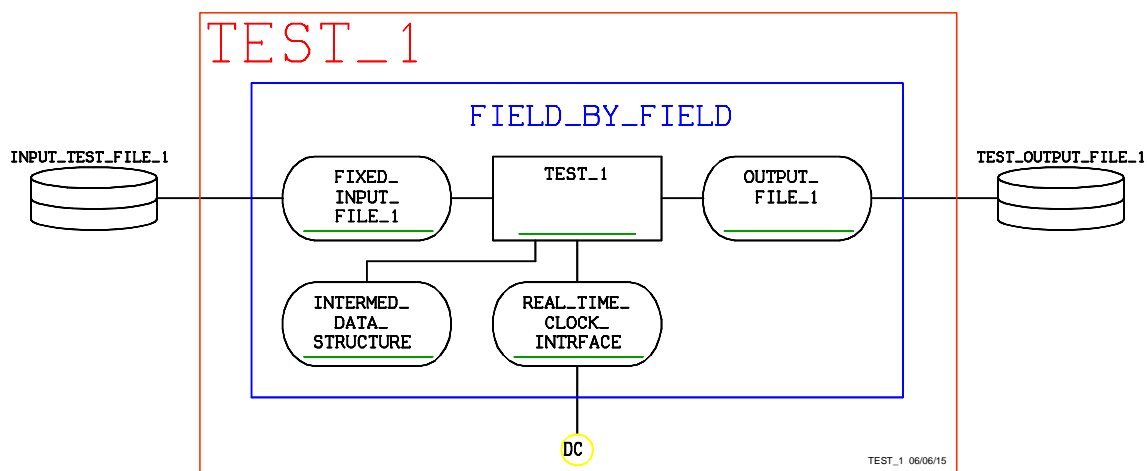


Figure 17-6. Test-1- Input File Record Formats.

Test 1 moves the file input records to the data structures shown in Figure 17-7(a). Note that the record can be moved into the resource structure. Because of the REDEFINES clause, the RECORD_TYPE can be checked and the matching record format (G or T) can be used directly.

Having moved the record into the input resource, the fields are transferred, FIELD_BY_FIELD, to the intermediate format shown in Figure 17-7(b). The DECIMAL fields are moved directly to the INTEGER fields with the exception that the intermediate triangle coordinates are put into REAL numbers as would be needed for computing the inner-product.

G_RECORD		
1	RECORD_TYPE	CHAR 1
1	TRIANGLE_NUMBER_G	DECIMAL 9(10)
1	LEVEL_NUMBER	DECIMAL 9
1	RIGHT_HAND_COLOR	DECIMAL 99
1	RIGHT_HAND_REFLECTION	DECIMAL 9
1	LEFT_HAND_COLOR	DECIMAL 99
1	LEFT_HAND_REFLECTION	DECIMAL 9
1	PADDING	CHAR 21
T_RECORD REDEFINES G_RECORD		
1	RECORD_TYPE	CHAR 1
1	TRIANGLE_NUMBER	DECIMAL 9(10)
1	VERTEX_NUMBER	DECIMAL 9
1	X_COORDINATE	DECIMAL 9(9)
1	Y_COORDINATE	DECIMAL 9(9)
1	Z_COORDINATE	DECIMAL 9(9)

Figure 17-7(a). Test-1- Input File Record Formats.

TRIANGLE_RECORD		
1	TRIANGLE_NUM	INTEGER
1	LEVEL_NUMBER	INTEGER
1	RIGHT_COLOR	INTEGER
1	RIGHT_REFLEC	INTEGER
1	LEFT_COLOR	INTEGER
1	LEFT_REFLEC	INTEGER
1	TRIANGLE_DATA	
2	VERTEX_1	
3	X_COORD_1	REAL
3	Y_COORD_1	REAL
3	Z_COORD_1	REAL
2	VERTEX_2	
3	X_COORD_2	REAL
3	Y_COORD_2	REAL
3	Z_COORD_2	REAL
2	VERTEX_3	
3	X_COORD_3	REAL
3	Y_COORD_3	REAL
3	Z_COORD_3	REAL

Figure 17-7(b). Test-1- Intermediate Data Structure Format.

Given that the inner-product could now be computed (it is not), the intermediate format is then converted back, FIELD_BY_FIELD, to the ASCII format shown in Figure 17-7(c) to write the records onto the new output file. In this test, we have produced the same output format as the input, with the exception of the INNER_PRODUCT. This “basic” approach gets changed in the following tests to improve speed.

G_RECORD_O		
1	RECORD_TYPE	CHAR 1
1	TRIANGLE_NUMBER_O	DECIMAL 9(10)
1	LEVEL_NUMBER_O	DECIMAL 9
1	RIGHT_HAND_COLOR_O	DECIMAL 99
1	RIGHT_HAND_REFLECTION_O	DECIMAL 9
1	LEFT_HAND_COLOR_O	DECIMAL 99
1	LEFT_HAND_REFLECTION_O	DECIMAL 9
1	INNER_PRODUCT_O	DECIMAL .999
1	PADDING_O	CHAR 17
T_RECORD_O REDEFINES G_RECORD_O		
1	RECORD_TYPE	CHAR 1
1	TRIANGLE_NUMBER_TO	DECIMAL 9(10)
1	VERTEX_NUMBER_O	DECIMAL 9
1	X_COORDINATE_O	DECIMAL 9(9)
1	Y_COORDINATE_O	DECIMAL 9(9)
1	Z_COORDINATE_O	DECIMAL 9(9)

Figure 17-7(c). Test-1- Output File Record Formats.

Test 2 Description

This test uses the Fixed Input File format specified in Figure 17-8(a), still an ASCII format, but instead of four separate records, a single record is used to represent the triangle information. Although the record size is larger, only one record is read instead of 4. We note the increased complexity of the hierarchy. Yet, when one compares Figure 17-8(a) to Figure 17-7(a), it is much easier to see the information on each triangle and its vertices along with the other graphical information. *The hierarchy makes it more understandable.*

TRIANGLE_RECORD_IN		
1	TRIANGLE_NUMBER_I	DECIMAL 9(10)
1	LEVEL_NUMBER_I	DECIMAL 9
1	RIGHT_HAND_COLOR_I	DECIMAL 99
1	RIGHT_HAND_REFLECTION_I	DECIMAL 9
1	LEFT_HAND_COLOR_I	DECIMAL 99
1	LEFT_HAND_REFLECTION_I	DECIMAL 9
1	TRIANGLE_DATA	
2	VERTEX_1	
3	X_COORD_1	DECIMAL 9(9)
3	Y_COORD_1	DECIMAL 9(9)
3	Z_COORD_1	DECIMAL 9(9)
2	VERTEX_2	
3	X_COORD_2	DECIMAL 9(9)
3	Y_COORD_2	DECIMAL 9(9)
3	Z_COORD_2	DECIMAL 9(9)
2	VERTEX_3	
3	X_COORD_3	DECIMAL 9(9)
3	Y_COORD_3	DECIMAL 9(9)
3	Z_COORD_3	DECIMAL 9(9)

Figure 17-8(a). Test-2- Input File Record Formats.

Having moved the record into the input resource, the fields are again transferred to the intermediate format shown in Figure 17-8(b). The DECIMAL fields are moved one-by-one to the INTEGER fields or to REAL numbers of the triangle coordinates for computing the inner-product.

TRIANGLE		
1	TRIANGLE_NUMBER_M	INTEGER
1	LEVEL_NUMBER_M	INDEX
1	RIGHT_HAND_COLOR_M	INDEX_1
1	RIGHT_HAND_REFLECTION_M	INDEX
1	LEFT_HAND_COLOR_M	INDEX_1
1	LEFT_HAND_REFLECTION_M	INDEX
1	TRIANGLE_M	
2	VERTEX_NUM_M1	
3	X_COORD_M1	REAL
3	Y_COORD_M1	REAL
3	Z_COORD_M1	REAL
2	VERTEX_NUM_M2	
3	X_COORD_M2	REAL
3	Y_COORD_M2	REAL
3	Z_COORD_M2	REAL
2	VERTEX_NUM_M3	
3	X_COORD_M3	REAL
3	Y_COORD_M3	REAL
3	Z_COORD_M3	REAL

Figure 17-8(b). Test-2- Intermediate Data Structure Format.

Without actually computing the inner-product, the intermediate format is then converted back to the ASCII format shown in Figure 17-8(c) to write the records onto the new output file.

TRIANGLE_RECORD_IN		
1	TRIANGLE_NUMBER_O	DECIMAL 9(10)
1	LEVEL_NUMBER_O	DECIMAL 9
1	RIGHT_HAND_COLOR_O	DECIMAL 99
1	RIGHT_HAND_REFLECTION_O	DECIMAL 9
1	LEFT_HAND_COLOR_O	DECIMAL 99
1	LEFT_HAND_REFLECTION_O	DECIMAL 9
1	INNER_PRODUCT_O	DECIMAL .999
1	TRIANGLE_DATA	
2	VERTEX_NUM_1	
3	X_COORD_O1	DECIMAL 9(9)
3	Y_COORD_O1	DECIMAL 9(9)
3	Z_COORD_O1	DECIMAL 9(9)
2	VERTEX_NUM_2	
3	X_COORD_O2	DECIMAL 9(9)
3	Y_COORD_O2	DECIMAL 9(9)
3	Z_COORD_O2	DECIMAL 9(9)
2	VERTEX_NUM_3	
3	X_COORD_O3	DECIMAL 9(9)
3	Y_COORD_O3	DECIMAL 9(9)
3	Z_COORD_O3	DECIMAL 9(9)

Figure 17-8(c). Test-2- Output File Record Formats.

Test 3 Description

This test uses a binary file format instead of ASCII, as shown in Figure 17-9(a). Using a binary file containing integers for T4 will cut the file size by more than half. In addition, there is no translation from ASCII numbers to binary in going to the intermediate format shown in Figure 17-9(b). Because of the different field types, they still must be moved individually.

TRIANGLE_RECORD_IN		
1	TRIANGLE_NUMBER_I	INTEGER
1	LEVEL_NUMBER_I	INTEGER
1	RIGHT_HAND_COLOR_I	INTEGER
1	RIGHT_HAND_REFLECTION_I	INTEGER
1	LEFT_HAND_COLOR_I	INTEGER
1	LEFT_HAND_REFLECTION_I	INTEGER
1	TRIANGLE_DATA	
2	VERTEX_NUM_1	
3	X_COORD_1	INTEGER
3	Y_COORD_1	INTEGER
3	Z_COORD_1	INTEGER
2	VERTEX_NUM_2	
3	X_COORD_2	INTEGER
3	Y_COORD_2	INTEGER
3	Z_COORD_2	INTEGER
2	VERTEX_NUM_3	
3	X_COORD_3	INTEGER
3	Y_COORD_3	INTEGER
3	Z_COORD_3	INTEGER

Figure 17-9(a). Test-3- Input File Record Formats.

TRIANGLE		
1	TRIANGLE_NUMBER_M	INTEGER
1	LEVEL_NUMBER_M	INDEX
1	RIGHT_HAND_COLOR_M	INDEX_1
1	RIGHT_HAND_REFLECTION_M	INDEX
1	LEFT_HAND_COLOR_M	INDEX_1
1	LEFT_HAND_REFLECTION_M	INDEX
1	TRIANGLE_M	
2	VERTEX_NUM_M1	
3	X_COORD_M1	REAL
3	Y_COORD_M1	REAL
3	Z_COORD_M1	REAL
2	VERTEX_NUM_M2	
3	X_COORD_M2	REAL
3	Y_COORD_M2	REAL
3	Z_COORD_M2	REAL
2	VERTEX_NUM_M3	
3	X_COORD_M3	REAL
3	Y_COORD_M3	REAL
3	Z_COORD_M3	REAL

Figure 17-9(b). Test-3- Intermediate Data Structure Format.

Given that the inner-product would be computed, the intermediate format is then converted back to the binary format shown in Figure 17-9(c), with the addition of the inner-product field. and the records written onto the new output file Figure 17-9(c). In this test, we have introduced binary file formats to improve speed.

TRIANGLE_RECORD_OUT		
1	TRIANGLE_NUMBER_O	INTEGER
1	LEVEL_NUMBER_O	INTEGER
1	RIGHT_HAND_COLOR_O	INTEGER
1	RIGHT_HAND_REFLECTION_O	INTEGER
1	LEFT_HAND_COLOR_O	INTEGER
1	LEFT_HAND_REFLECTION_O	INTEGER
1	INNER_PRODUCT_O	REAL
1	TRIANGLE_DATA	
2	VERTEX_1	
3	X_COORD_O1	INTEGER
3	Y_COORD_O1	INTEGER
3	Z_COORD_O1	INTEGER
2	VERTEX_2	
3	X_COORD_O2	INTEGER
3	Y_COORD_O2	INTEGER
3	Z_COORD_O2	INTEGER
2	VERTEX_3	
3	X_COORD_O3	INTEGER
3	Y_COORD_O3	INTEGER
3	Z_COORD_O3	INTEGER

Figure 17-9(c). Test-3- Output File Record Formats.

Test 4 Description

Test 4 is similar to Test 3, except this test blocks the file, putting 800 records into a block. This eliminates the need to read each record from disk. It also increases the level of the hierarchy. But the blocked records are easy to denote. After reading a block, the records are moved individually to the intermediate format.

```

TRIANGLE_BLOCK_OUT
  1 TRIANGLE_RECORD    QUANTITY(800)
    2 TRIANGLE_NUMBER_I      INTEGER
    2 LEVEL_NUMBER_I         INTEGER
    2 RIGHT_HAND_COLOR_I     INTEGER
    2 RIGHT_HAND_REFLECTION_I INTEGER
    2 LEFT_HAND_COLOR_I      INTEGER
    2 LEFT_HAND_REFLECTION_I  INTEGER
    2 TRIANGLE_DATA
      3 VERTEX_NUM_1
        4 X_COORD_1      INTEGER
        4 Y_COORD_1      INTEGER
        4 Z_COORD_1      INTEGER
      3 VERTEX_NUM_2
        4 X_COORD_2      INTEGER
        4 Y_COORD_2      INTEGER
        4 Z_COORD_2      INTEGER
      3 VERTEX_NUM_3
        4 X_COORD_3      INTEGER
        4 Y_COORD_3      INTEGER
        4 Z_COORD_3      INTEGER

```

Figure 17-10(a). Test-4- Input File Record Formats.

```

TRIANGLE
  1 TRIANGLE_NUMBER_M      INTEGER
  1 LEVEL_NUMBER_M         INDEX
  1 RIGHT_HAND_COLOR_M     INDEX_1
  1 RIGHT_HAND_REFLECTION_M INDEX
  1 LEFT_HAND_COLOR_M      INDEX_1
  1 LEFT_HAND_REFLECTION_M INDEX
  1 TRIANGLE_M
    2 VERTEX_NUM_M1
      3 X_COORD_M1      REAL
      3 Y_COORD_M1      REAL
      3 Z_COORD_M1      REAL
    2 VERTEX_NUM_M2
      3 X_COORD_M2      REAL
      3 Y_COORD_M2      REAL
      3 Z_COORD_M2      REAL
    2 VERTEX_NUM_M3
      3 X_COORD_M3      REAL
      3 Y_COORD_M3      REAL
      3 Z_COORD_M3      REAL

```

Figure 17-10(b). Test-4- Intermediate Data Structure Format.

Assuming that the inner-product was computed, the intermediate format is then converted back to the binary format shown in Figure 17-10(c) to write the 800 record blocks onto the new output file. Again, in this test, we have produced the same output format as the input, with the exception of the INNER_PRODUCT.

TRIANGLE_BLOCK_OUT		
1	TRIANGLE_RECORD	QUANTITY(800)
2	INNER_PRODUCT	REAL
2	REST_OF_RECORD	
3	TRIANGLE_NUMBER_O	INTEGER
3	LEVEL_NUMBER_O	INTEGER
3	RIGHT_HAND_COLOR_O	INTEGER
3	RIGHT_HAND_REFLECTION_O	INTEGER
3	LEFT_HAND_COLOR_O	INTEGER
3	LEFT_HAND_REFLECTION_O	INTEGER
3	TRIANGLE_DATA	
4	VERTEX_1	
5	X_COORD_O1	INTEGER
5	Y_COORD_O1	INTEGER
5	Z_COORD_O1	INTEGER
4	VERTEX_2	
5	X_COORD_O2	INTEGER
5	Y_COORD_O2	INTEGER
5	Z_COORD_O2	INTEGER
4	VERTEX_3	
5	X_COORD_O3	INTEGER
5	Y_COORD_O3	INTEGER
5	Z_COORD_O3	INTEGER

Figure 17-10(c). Test-4- Output File Record Formats.

Test 5 Description

This test is similar to Test 4 using blocked files. However, it uses REAL numbers on the file instead of integers. This eliminates the need to perform number conversions from INTEGER to REAL, allowing group moves of the binary numbers. Given that the file contains REAL numbers, the entire triangle can be moved since the fields all match.. This cuts the movement of numbers by a factor of 9. In addition, the other triangle graphical attributes are designated precisely, where INDEX is a 2-byte integer, and INDEX_1 is a 1-byte integer.

```

TRIANGLE_BLOCK_IN
  1 TRIANGLE_RECORD_I    QUANTITY(800)
  2 TRIANGLE_NUMBER_I    INTEGER
  2 LEVEL_NUMBER_I        INDEX
  2 RIGHT_HAND_COLOR_I    INDEX
  2 RIGHT_HAND_REFLECTION_I INDEX_1
  2 LEFT_HAND_COLOR_I     INDEX
  2 LEFT_HAND_REFLECTION_I INDEX_1
  2 TRIANGLE_DATA
    3 VERTEX_NUM_1
      4 X_COORD_1        REAL
      4 Y_COORD_1        REAL
      4 Z_COORD_1        REAL
    3 VERTEX_NUM_2
      4 X_COORD_2        REAL
      4 Y_COORD_2        REAL
      4 Z_COORD_2        REAL
    3 VERTEX_NUM_3
      4 X_COORD_3        REAL
      4 Y_COORD_3        REAL
      4 Z_COORD_3        REAL

```

Figure 17-11(a). Test-5- Input File Record Formats.

```

TRIANGLE
  1 TRIANGLE_NUMBER_M    INTEGER
  1 LEVEL_NUMBER_M        INDEX
  1 RIGHT_HAND_COLOR_M    INDEX
  1 RIGHT_HAND_REFLECTION_M INDEX_1
  1 LEFT_HAND_COLOR_M     INDEX
  1 LEFT_HAND_REFLECTION_M INDEX_1
  1 TRIANGLE_M
    2 VERTEX_NUM_M1
      3 X_COORD_M1        REAL
      3 Y_COORD_M1        REAL
      3 Z_COORD_M1        REAL
    2 VERTEX_NUM_M2
      3 X_COORD_M2        REAL
      3 Y_COORD_M2        REAL
      3 Z_COORD_M2        REAL
    2 VERTEX_NUM_M3
      3 X_COORD_M3        REAL
      3 Y_COORD_M3        REAL
      3 Z_COORD_M3        REAL

```

Figure 17-11(b). Test-5- Intermediate Data Structure Format.

Again, assuming that the inner-product was computed, the intermediate format is then moved back to the binary format shown in Figure 17-10(c) to write the 800 record blocks onto the new output file. Again, in this test, we have produced the same output format as the input, with the exception of the INNER_PRODUCT. However, records may be moved with a single instruction fetch, since the rest of the hierarchies match.

TRIANGLE_BLOCK_OUT		
1	TRIANGLE_RECORD	QUANTITY(800)
2	INNER_PRODUCT	REAL
2	REST_OF_RECORD	
3	TRIANGLE_NUMBER_O	INTEGER
3	LEVEL_NUMBER_O	INDEX
3	RIGHT_HAND_COLOR_O	INDEX
3	RIGHT_HAND_REFLECTION_O	INDEX_1
3	LEFT_HAND_COLOR_O	INDEX
3	LEFT_HAND_REFLECTION_O	INDEX_1
3	TRIANGLE_DATA	
4	VERTEX_1	
5	X_COORD_O1	REAL
5	Y_COORD_O1	REAL
5	Z_COORD_O1	REAL
4	VERTEX_2	
5	X_COORD_O2	REAL
5	Y_COORD_O2	REAL
5	Z_COORD_O2	REAL
4	VERTEX_3	
5	X_COORD_O3	REAL
5	Y_COORD_O3	REAL
5	Z_COORD_O3	REAL

Figure 17-11(c). Test-5- Output File Record Formats.

SUMMARY OF TEST RESULTS

The test results of the five approaches are shown in Table 17-1.

Table 17-1. Results of File Format Tests (Seconds)

Test #	# of Triangles	Time 1	Time 2	Time 3	Average Time
1	1,000,000	74.25	76.03	78.44	76.24
2	1,000,000	50.68	52.31	51.91	51.63
3	1,000,000	6.88	6.61	5.35	6.28
4	1,000,000	1.75	1.76	2.3	1.94
5	1,000,000	0.70	0.75	0.78	0.74

The difference in times from the first test to the last is a factor of 103. Simply put, the 5th approach is over 100 times faster than the 1st approach. A plot of the results is shown in Figure 17-12.

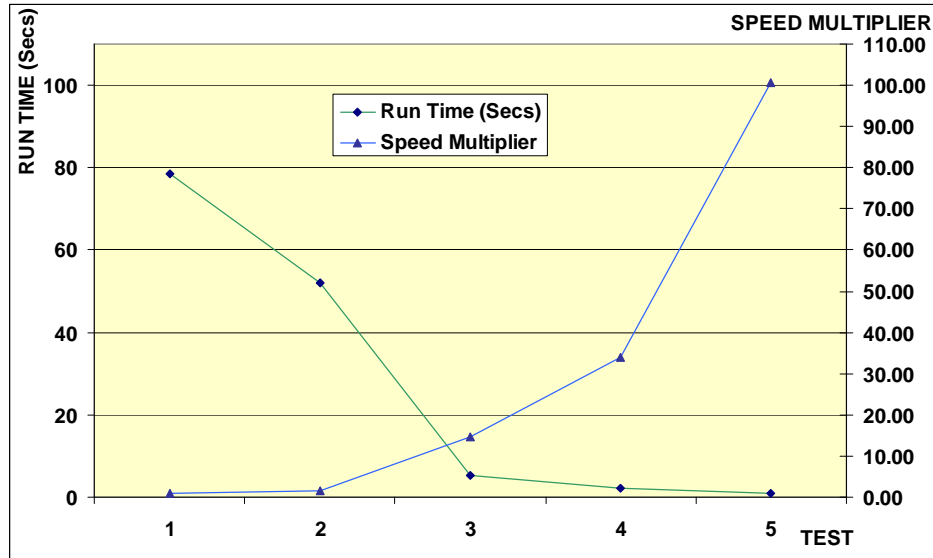


Figure 17-12. Plot of test results.

Validation Of Results

To ensure that the output file contained the correct information, separate tasks were built for tests 3, 4, and 5 to create a print file with a readable format. This allowed for review of the resulting binary files to ensure that the data was correct.

Analysis Of Results

Clearly, the design of the data structures within the records as well as the intermediary resource is critical to run-time speed. The factors addressed in the tests are the following:

- **Hierarchical Structures** - As the hierarchical structures became deeper in each test case above, the movement of data was reduced and speed improved. At the same time, as deeper hierarchies were introduced, the VisiSoft code became much more simple and easily understood. This is because the data is better organized.
- **Matching Structures** - The use of matching data structures allows one to perform group moves. This is most easily accomplished using hierarchical structures.
- **Data Types** - The selection of data types, e.g., ASCII (Text) numbers versus binary numbers directly affects the need to perform transformations to match the internal binary coding.
- **File Formats** - The increased hierarchy and file formatting directly affects the speed with which files may be read, as well as the need to perform transformations to match the internal binary coding.
- **Record Blocking** - The use of hierarchical record blocking directly affects the speed of reading large files. When records are blocked, large numbers may be read with a single read statement. Internal use of record counters is fast.

Speed Test Results Are Independent Of Architecture

It is apparent from the above tests that the architecture had no effect on speed (the architecture is identical for every test). The deciding factor was the ease with which one can build hierarchical structures with data type definitions that are easily understood - clearly a function of the language. The main result of these tests is that it is the language that determined run-time speed, being the major factor affecting the understandability of how to achieve the huge improvements speed.

We must emphasize that we have Grace Hopper to thank for the types of data structures contained in VisiSoft. Although there are more facilities in VisiSoft that make it easier to use than COBOL or CMS-2, the basic concepts came from her insights into language design. We know of no one who has come close to her knowledge in this area. These concepts apply directly to the instruction code (VisiSoft Processes) where the hierarchical facilities simplify the understanding of extremely complex algorithms - so they are easily understood by a subject area expert. Looking back, it was the data processing departments in large corporations that were under the gun to process large transaction and customer account files under tight deadlines. The scientific community is scarcely ever under that gun.

EXPERIMENTATION IS KEY TO UNDERSTANDING DIFFERENCES

Experimenters reading this chapter are encouraged to rewrite the above tests in C-based languages. Again, higher speeds can be achieved using hierarchical data structures. However, understandability of these structures quickly becomes difficult as the layers in the hierarchy are increased. Worse is the level of complexity of the resulting instruction code that uses these structures. Because of the increased organization of the data using the language facilities in VisiSoft, understandability is clearly increased with algorithm simplification - along with speed.

Obviously one is going to get different speeds from these tests when running them on different computers. However, it is the relative (normalized) speed differences that are of interest when trying to draw conclusions on how to make software run faster.

Although the tests described above were limited to reading and writing large files, similar tests can be run to determine the effects on internal processing. These can be done for data scanning and manipulation, or mathematical algorithms. To get sufficiently valid results, one typically has to run large sequences of cases so that reasonable times are used for comparison. Certain of the authors have performed such tests. They confirm the results described above with regard to hierarchical data structures. This is particularly true when dealing with complex databases where the hierarchical structures can be quite large. Readers are encouraged to conduct such tests on their own to draw and evaluate their own conclusions.

CHAPTER 18.

PARALLEL PROCESSOR TESTS & RESULTS

OVERVIEW OF EXPERIMENTS

This chapter provides more samples of the kinds of experiments and testing that have been performed by the authors. It contains a comparison of results run on a parallel processor. The Windows “oscilloscope” is used in the first experiment, backed up in the second experiment by times off the real-time clock to interpret the results. The purpose of these experiments is to compare measured data to the theoretical expectations of the authors’ designs. As in the prior chapter, the comparisons are obvious.

A SET OF PARALLEL PROCESSOR EXPERIMENTS

The first simulation was built to test the initial version of the Run-Time System (RTS) and the VisiSoft Parallel Operating System (VPOS). The architecture for this simulation is shown in Figure 18-1. Note that this simulation has six Independent (IND) modules that are capable of running either as a single simulation on one processor, or as a parallel processor simulation on VSI’s Parallel PC, AKA the PPC.

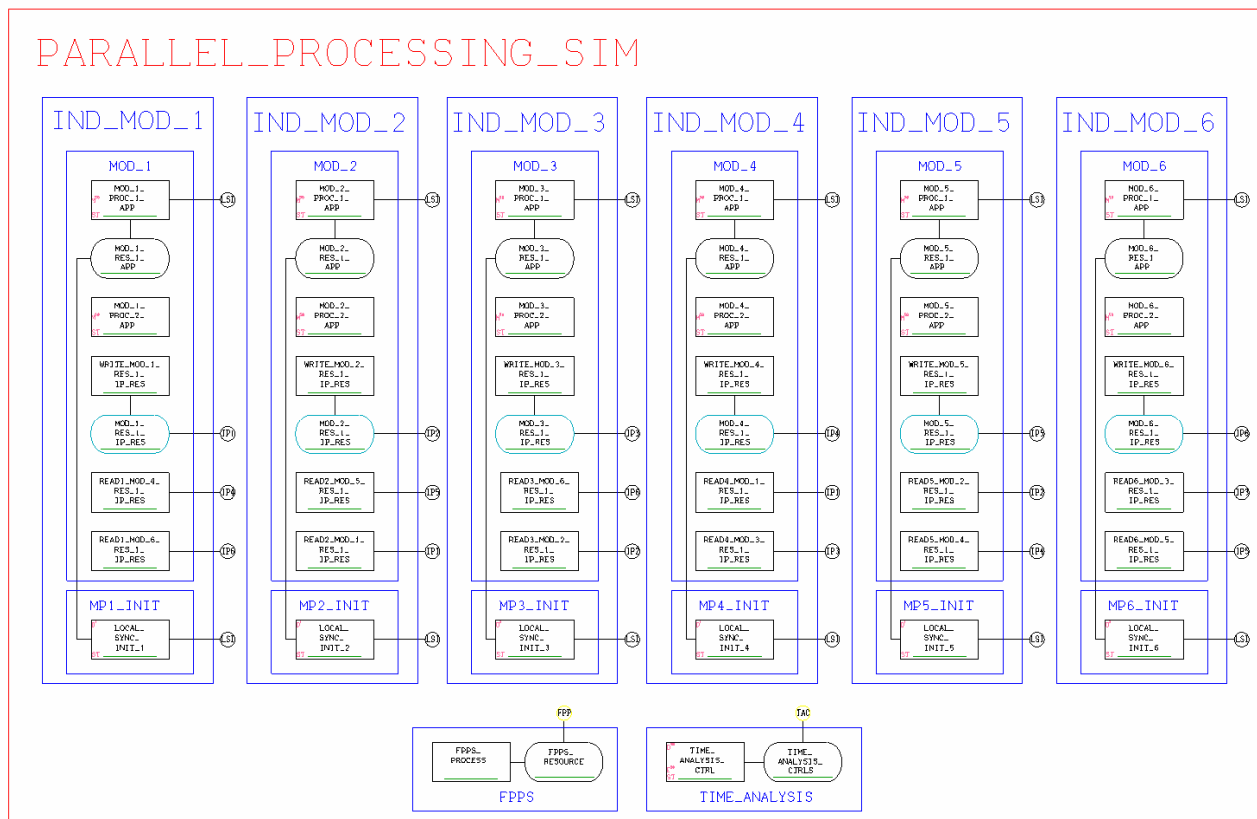


Figure 18-1 – Architecture of Simulation running on the PPC

Key elements of this test simulation are the following:

1. It replicates a typical large scale simulation with a reasonably high degree of inherent parallelism. As such, the IND modules communicate among each other at critical points in time, and their simulation clocks must remain synchronized within a predefined ΔT_{\max} time of each other. If this were not the case, they could be classified as embarrassingly parallel vs. inherently parallel.
2. In the example presented here, each IND module runs for 20 seconds of simulated time and the processes within each IND Module reschedule themselves once per second. In addition, at various times they *cross schedule* processes in other IND modules. This represents the real-world case, e.g., in communications networks, whereby most of the processing occurs within an IND module, but it must exchange information with other IND modules for its own processing. The frequency and quantity of information transfers can be varied while the simulation is running.
3. Each time a Process runs, it calls a subsystem representing a real-world load, which can be varied. In this way, one can study the overhead of cross schedules, communications, and synchronization among processors to remain within the ΔT_{\max} time constraint.
4. When run as a single simulation, the same number of schedules is made, the same subsystem processing load is used, and since all 6 IND modules are running on a single processor, cross schedules revert to normal schedules. This provides for the exact same model processing load whether run as a single simulation or on the PPC. Because the PPC incurs the parallel processor overhead components as noted above, this facility provides for an analysis of processing efficiency and scalability.
5. When run on the PPC, IND modules are assigned to Processors 1-6, the master VPOS is assigned to Processor 7, and Processor 8 is left for Run-Time Graphics. When running on a single processor, there is no attempt to assign the processor.

TEST RESULTS

Single Simulation Test Results

The single simulation was run multiple times, with an average running time of 108.4 seconds. Looking ahead to the PPC runs below, one could infer that if the PPC were perfectly scalable, the simulation should run in 18.07 (108.4/6) seconds on the PPC. The Windows Performance graph from one of these sample runs is shown in Figure 18-2, where an interesting outcome is consistently observed. For the single processor run shown, the simulation begins on Processor 2 (P2). Then at a point part way thru the simulation, Win-7 apparently decides to move the simulation to P1, only to immediately return it to P2. Then it is moved to P5 and P7, and finally back to P2 where it started. We note that in this test there is no attempt by VPOS to assign the simulation to a specific processor.

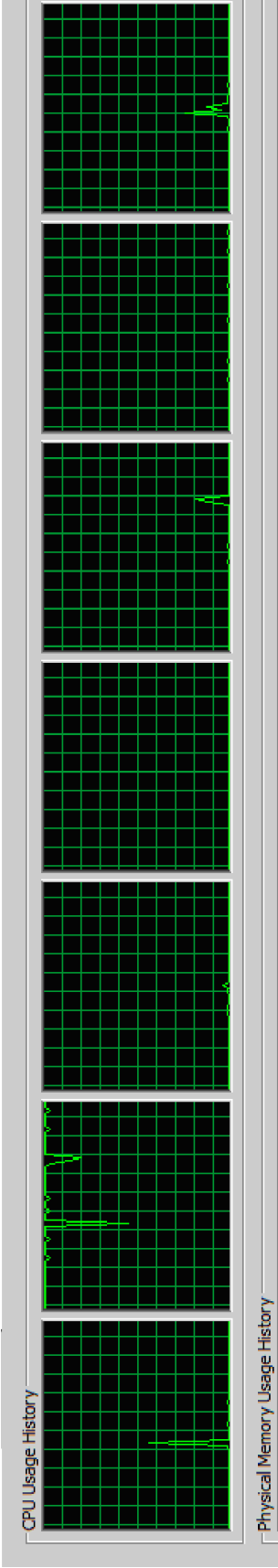


Figure 18-2. Single Simulation - Win-7 Processor Allocation.

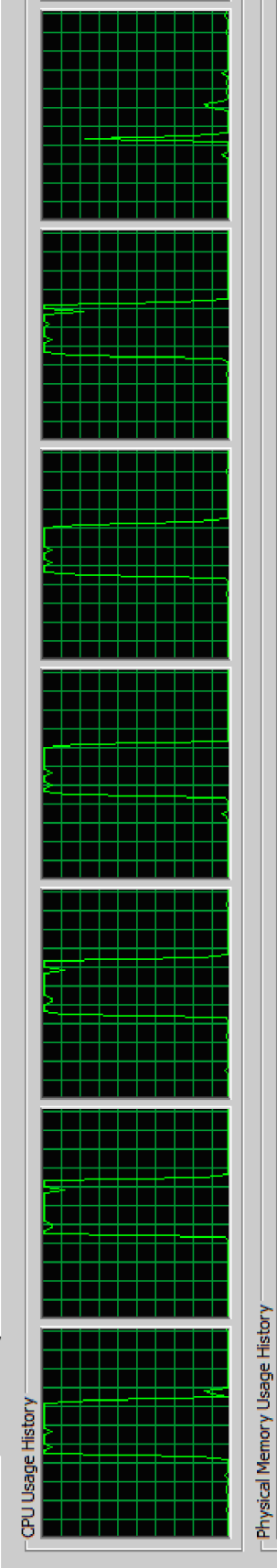


Figure 18-3. PPC Simulation - VPOS 7 Processor Allocation with no interval.

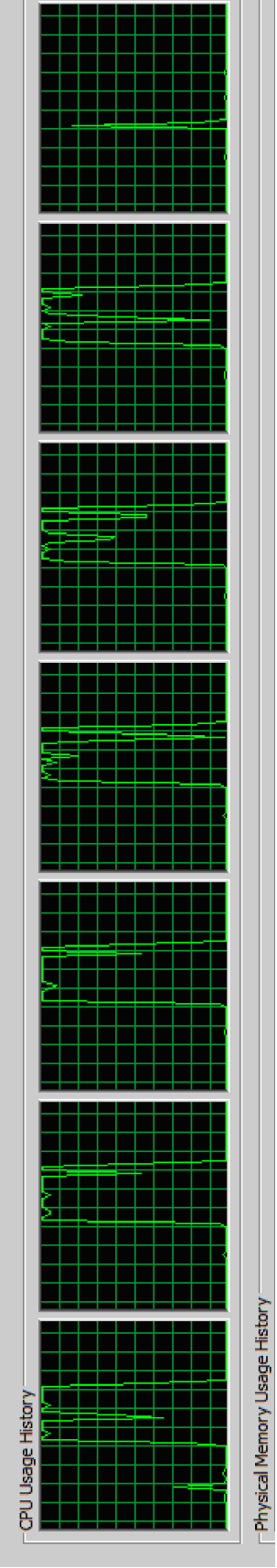


Figure 18-4. PPC Simulation - VPOS 7 Processor Allocation with tight interval.

Parallel Processor (PPC) Test 1 Results

A sample result from a set of PPC runs is shown in Figure 18-3. In these runs, the ΔT_{\max} interval was set outside the total run time of the simulation, allowing a closer look at the potential performance of a dedicated (minimal Windows contention) VPOS environment. Using a perfectly balanced 6-processor model load, the actual run-time result was 16.17 seconds. This result is interesting compared with the perfectly scaled projection of 18.07 seconds from the single processor simulation above.

Whereas the IND modules and VPOS Master Synchronizer are specifically assigned to processors as explained above, there was no attempt to corral the background Windows processes (in excess of 40) onto a single processor. This could explain the lag on P2 in Figure 18-2. A detailed design to eliminate this unwarranted overhead has already been produced for VPOS.

Parallel Processor (PPC) Test 2 Results

In the test shown in Figures 18-4, the Delta Time Interval was set to 8. This means that all six processors must synchronize with each other every 8 seconds, i.e., at simulation times 8, 16, 24, etc. The first processor to arrive at a “synch point” must wait the longest for the last processor to arrive. This can clearly be seen in Figure 18-4. All the processors among the first six except for P2 had some dip in cpu usage as they entered idle states of various duration waiting for P2 to arrive. P2 had no reason to go idle because, as soon as it arrived at the sync point, processing into the next Delta Time interval (sim time 8+ thru 16) resumed on all processors. It should be noted that these graphs correlate directly with the actual run times of each processor. They also correlate to the theoretical examples described in Chapter 6, and particularly the example shown in Figure 6-2.

The test results taken from one of the six processors show that the actual processing time was 18.39 seconds compared to the perfectly scaled projection of 18.07 seconds from the single processor simulation, above. The idle times shown in Figure 18-4 suggest that this difference is due to most of the background Windows processes being assigned to the lagging processor. When the simulation model load is identical across all processors (as is the case with these tests), the multiprocessor overhead is expected to be negligible. For real-world simulations, the normalized difference between single and PPC runs will be mostly due to model load imbalance.

Parallel Processor (PPC) Observations

We note that the VPOS implementation is being done in two stages. The tests described above were done on an 8 processor PC with the initial version of VPOS that allows the WINUX OS to still have some control of the processors. The next version of VPOS will remove this control, fielding hardware events and totally managing memory. For various reasons, we expect the new version of VPOS to run faster as well as remove the idiosyncrasies observed in these tests.

A MILITARY OPERATIONS PLANNING APPLICATION

Figure 18-5 provides an accurate representation of the Global Positioning System (GPS). As these satellites move in orbits around the globe, the ability to be connected to a particular satellite from a spot on the earth changes with Time Of Day (TOD).

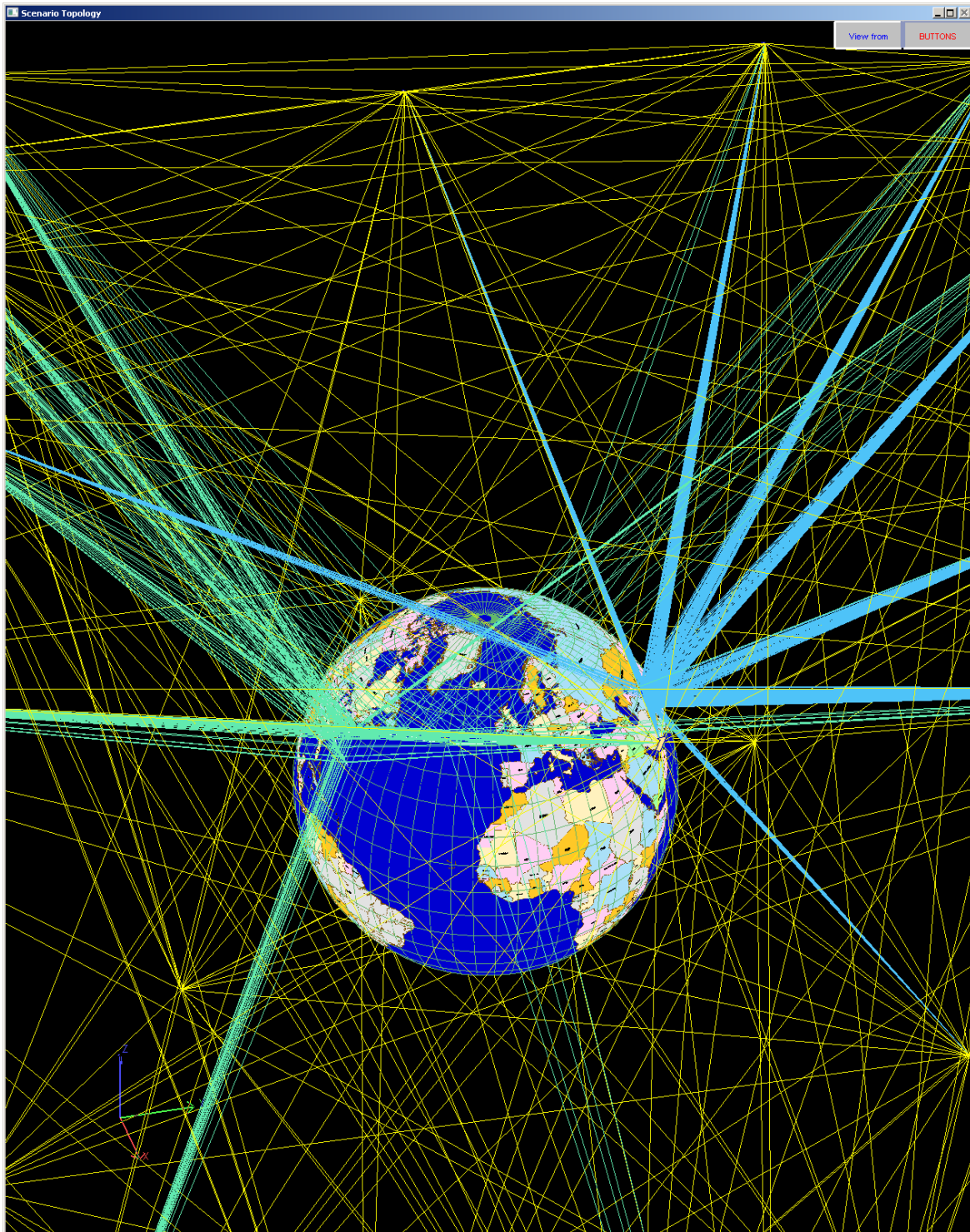


Figure 18-5. GPS satellite constellation with connectivity to air, sea, and ground platforms .

GPS is just one of many systems modeled in this planning application. To maintain accuracy of position, a sufficient number of satellites (typically 5 - 7) must be connected via RF signal. Planning military operations involves coordination of many systems. Some of these are described below to provide a feel for the processing problem, and the amount of software required to model these systems with sufficient accuracy. The simulation architecture follows the physical organization of these systems since that provides maximum use of inherent parallelism as well as the best correlation with actual operations.

Planning Dynamic Operations

Figure 18-6 provides an illustration of dynamic operations to be analyzed. One must be able to represent the movement and equipment operations of hundreds of platforms, interacting according to a plan, but being susceptible to potentially significant changes based upon the actions of other entities in the operation. To be valid, these actions and reactions must be based upon detailed models of the elements and sub-elements of the entities.

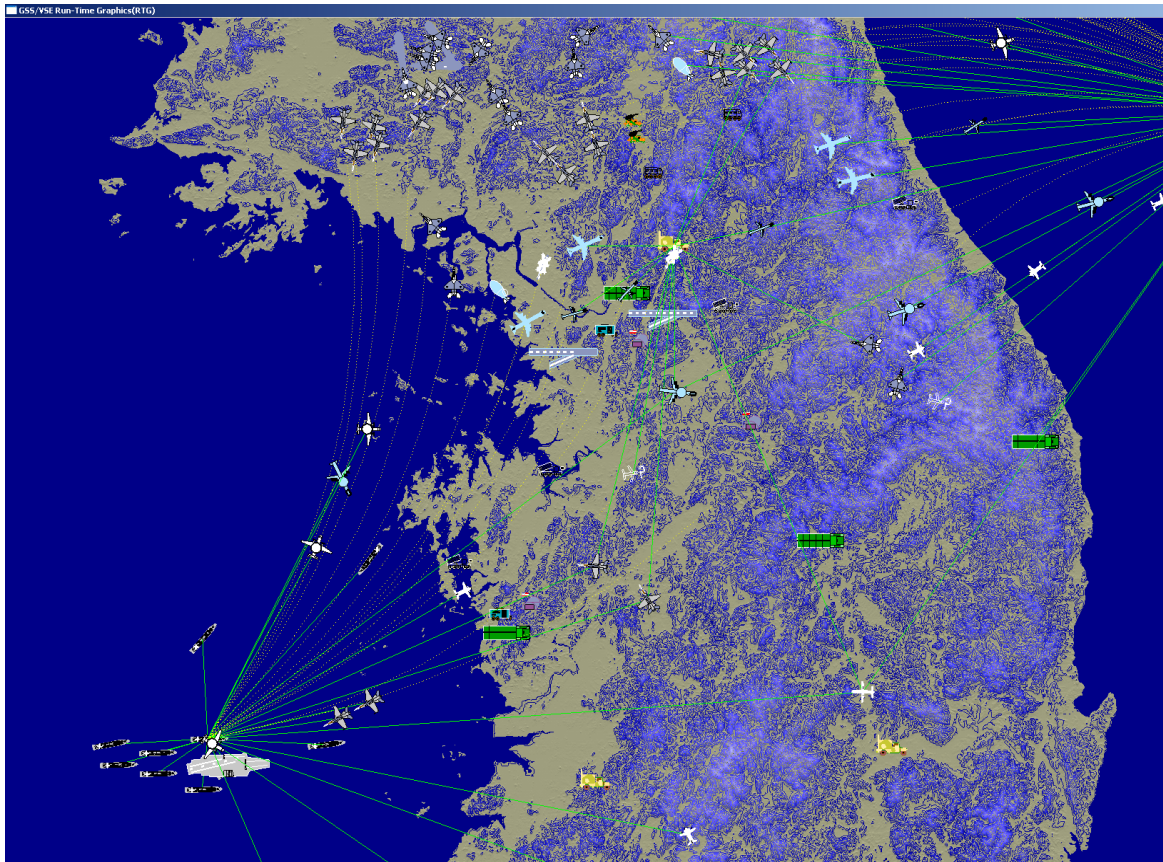


Figure 18-6. Example of dynamic operations to be analyzed.

In addition to the requirement for simulations to run very fast, there are requirements on the speed with which one can modify the scenarios, and the speed with which one can perform analysis. In both cases, the use of 3D interactive graphics is crucial to meeting these needs. Different simulations are used for planning and analysis so that users can interact with the running simulations. When performing multiple simulation runs for parametric analysis, the graphical outputs are still available and are designed to not slow down the simulations.

Model Architectures

Figure 18-7 provides a breakout of the types of models that must reside within this type of simulation to support various analyses. The basic IND modules in this simulation are Platform models, including ground, sea, air and space. All of these platform models contain various Equipment models, e.g., computers, radios, sensors, weapons, etc. Platforms interact via different environments using different equipment, e.g., sensors, radios, etc.

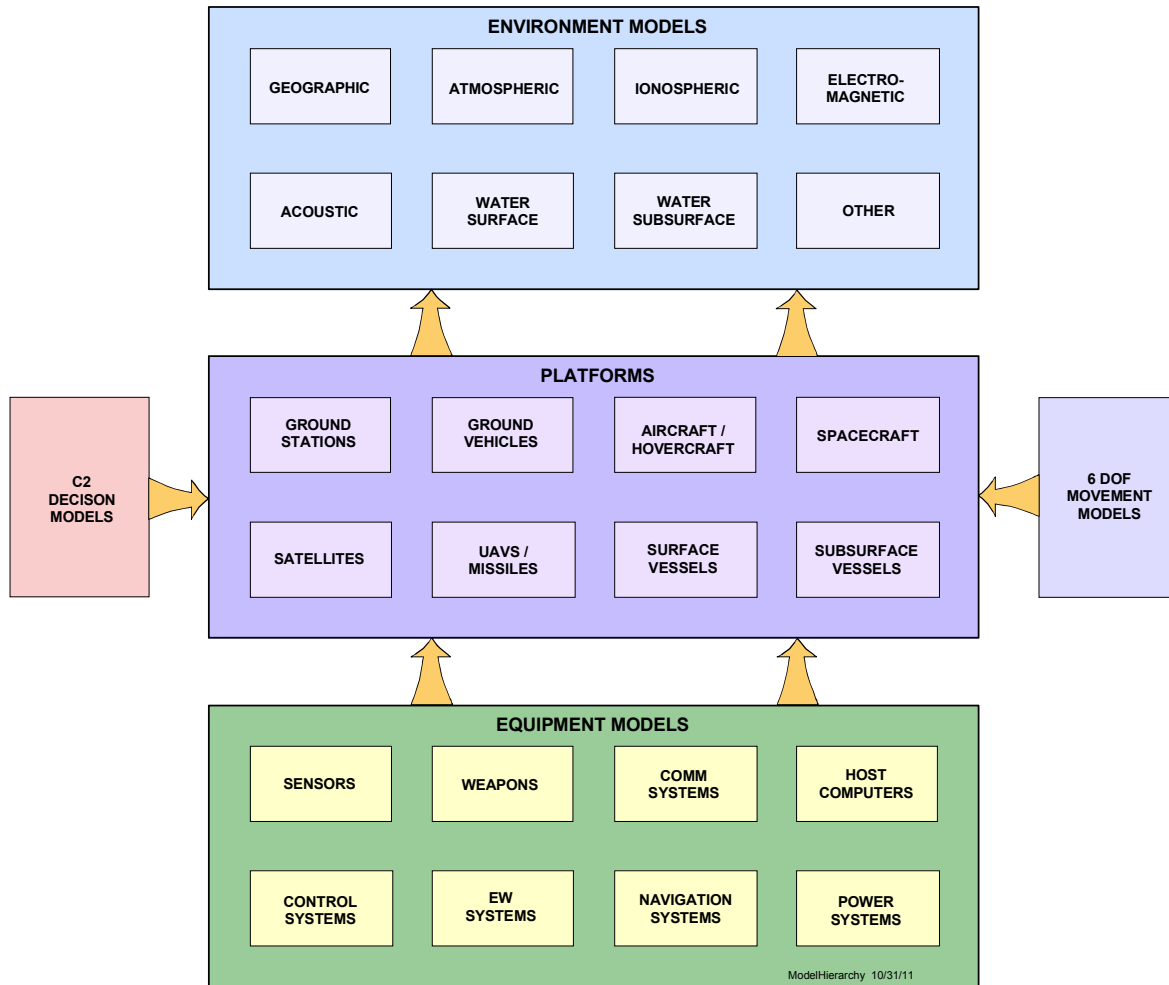


Figure 18-7. Overall architecture of the models.

Underlying this modeling facility is the software architecture technology that has evolved as a CAD facility for simulation since 1982. This CAD facility was designed to support the rapid development of models and simulations by subject area experts without help from programmers. The technology behind this facility has dramatically increased productivity, including ease of reuse and modification of models. This facility provides for selecting specific types of models by clicking on check boxes in a user-friendly panel. As an example, these include electro-magnetic wave models that account for 3D terrain, foliage, and suburban buildings.

Overall Simulation Architecture

The simulation architecture allows analysts to select those platforms to be incorporated into a simulation, as well as the equipment on each platform for rapid creation of complex scenarios. Figure 18-8 provides an illustration of a single processor simulation containing different platforms, each carrying different equipment. The simulation architecture provides for automatically integrating different equipment into specified platforms, and platforms into specified mission environments. Missions are defined by mission threads, including the movement of all platforms as well as sensor sightings and targeting events. Events are defined by decision tables in C2 models as well as messages, and all may be defined by input files including the movement paths to be used by each platform.

Although this architecture may be built on a single processor, it contains IP resources that interface between platforms. IP resources may be used on a single processor as well as a parallel processor, automatically taking on a different protocol when operating on a parallel processor.

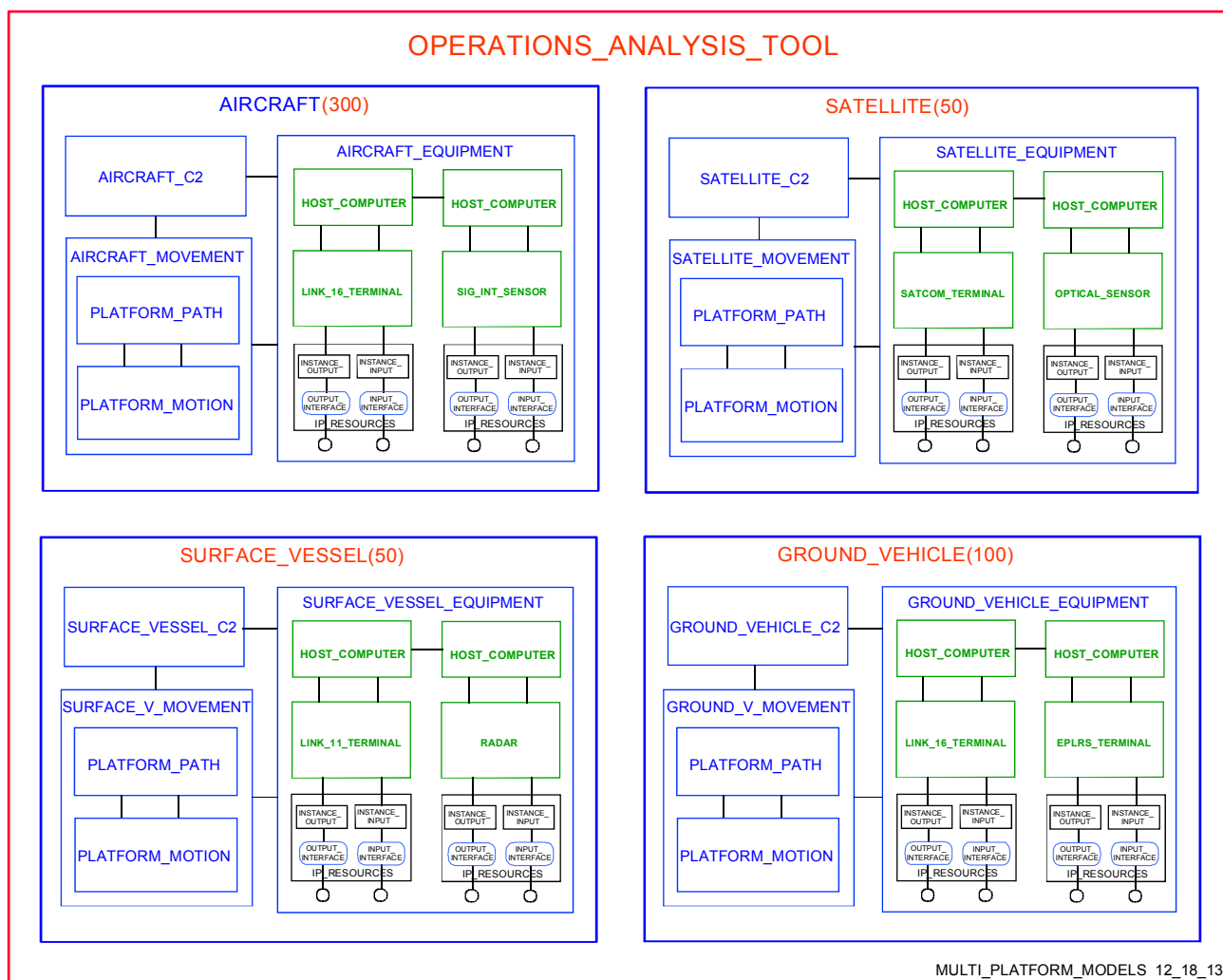


Figure 18-8. Overall simulation architecture.

Complex Scenario Creation & Modification

Figure 18-9 illustrates a graphical interface facility for interactively creating paths for air, sea, and ground platforms. Once these paths are created in terms of pathpoints, more detailed waypoints and movepoints are generated automatically. This illustrates the level of complexity of the models used in this simulation.

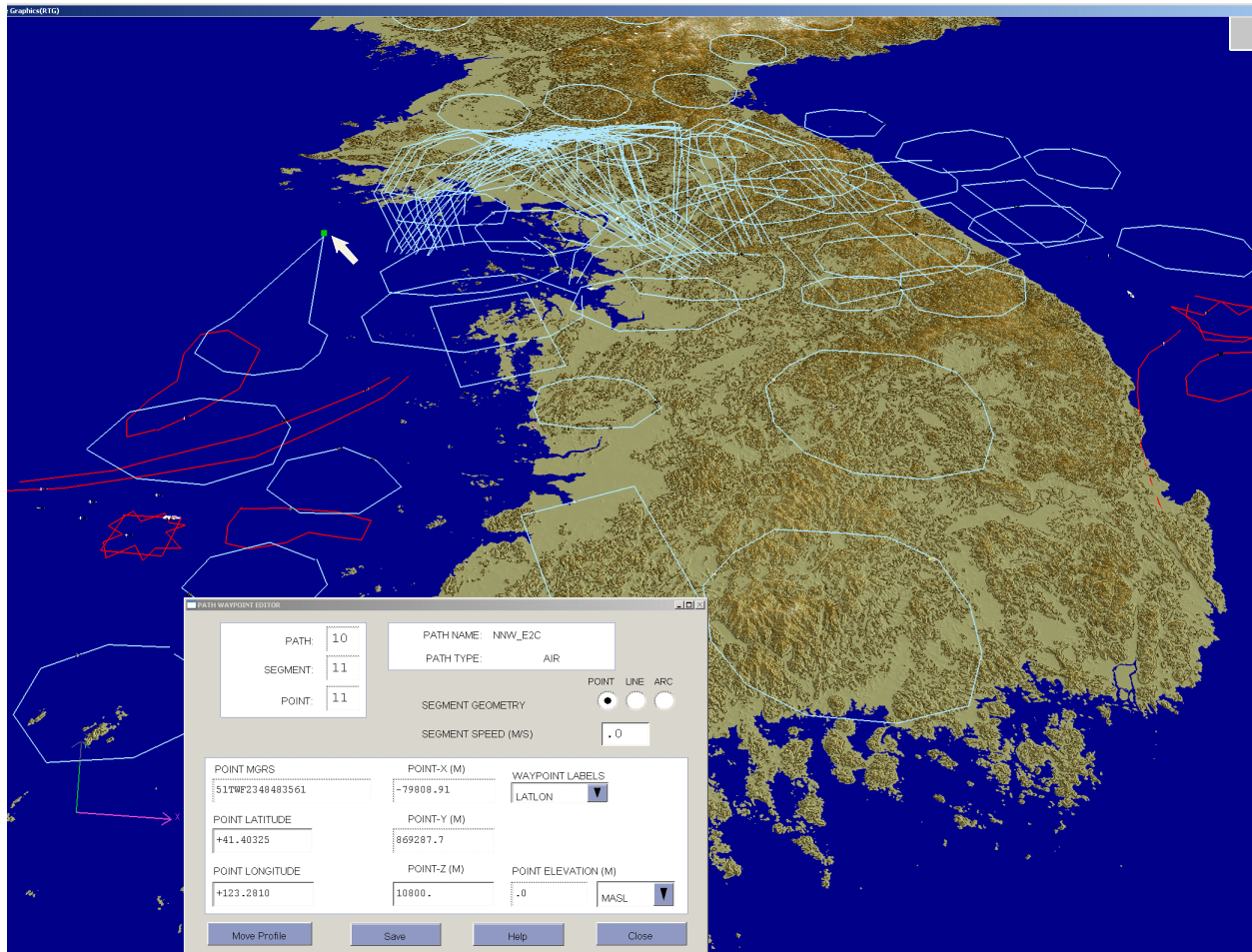


Figure 18-9. Illustration of the complexity of scenarios.

Figure 18-10 illustrates the connectivity between an air platform and platforms on the ground. Green lines indicate sufficient connectivity to receive messages, while red indicates lack of connectivity. These calculations depend upon large terrain databases. This screen shot is taken from a scenario in North-West Afghanistan.

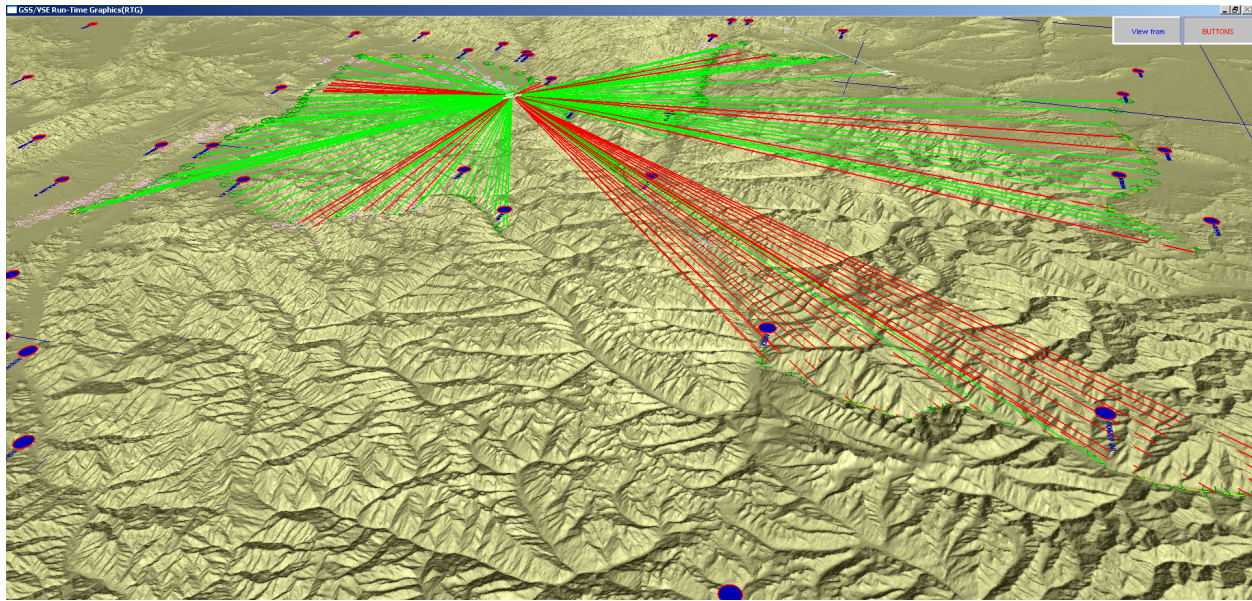


Figure 18-10. Illustration of propagation modeling to determine connectivity.

GLOBAL_PLANNER EXPERIMENT

This experiment uses the GLOBAL_PLANNER simulation which contains a sampling of models from many military simulations previously developed. The architecture for this simulation has been designed to take advantage of a parallel processor. A simplified version is shown in Figure 18-11. As in all simulations developed for parallel processors using this CAD system, it may be run on any number of processors with the exception that if the number of processors exceeds the number of IND modules + 2, they will not be used. It can be run on a single processor to allow single processor speed testing and comparison of results on multiple processors.

The version of GLOBAL_PLANNER used for testing has 10 IND modules of which 6 are shown in Figure 18-11. The first module, IND_MAIN supports model initialization, file inputs, and run-time outputs, including 3D Open-GL graphics. This module interfaces with processors in the server environment to handle file I/O and graphical interfaces. The second IND module, IND_SATELLITE, provides the GPS satellite constellation and communications to the other platforms on separate IND modules. The remaining modules contain air and sea platforms that provide tactical aircraft and ships that interact with the GPS satellites as well as each other. Each of these modules provides multiple platforms on multiple paths. Since VPOS and RTG require their own processors, a total of twelve processors are used to run this simulation.

The arched light green lines in Figure 18-11 indicate CALL statements in processes that invoke other processes within the same IND module. The arched red lines indicate statements in processes in one IND module that cross SCHEDULE processes within different IND modules. Showing these lines is an option in VisiSoft. As described in Section 12.3.4, the SCHEDULE statement is used to start threads which may reside in another IND module.

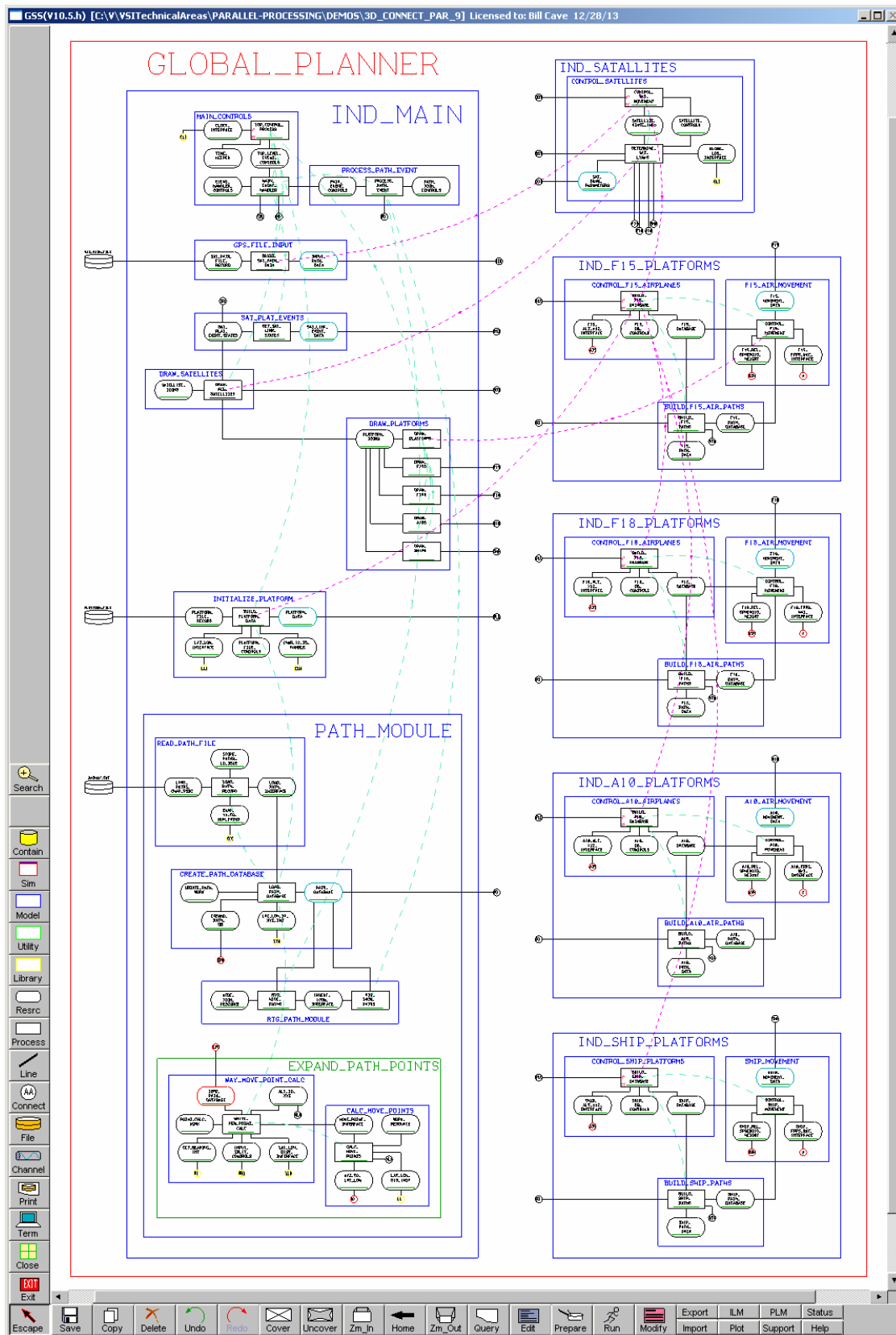


Figure 18-11. GLOBAL_PLANNER - a parallel processor simulation.

EXPERIMENT DESIGN FACTORS

The principle concern in this experiment is to produce sufficient data regarding the effects of specific application software design parameters on parallel processor speed. These include using different numbers of processors and different scenario segments. All of the possible factors affecting the evaluation of parallel processor speeds would take much more space than allotted here. Because of limited space in this theory book, the factors analyzed here are considered to be important relative to what is currently available. Much more is available to the interested reader and we look forward to reviewing the details of various experiments by those who desire to do so. Important variational factors observed in this experiment are described below.

Single Processor Versus Parallel Processor Speeds

The most important observations addressed here are the Parallel Processor Speed Multipliers (SMs) and Processor Utilization Efficiencies (PUEs) obtained when varying specific factors affecting speed on a single processor versus a parallel processor. The observations used to determine the effects on the application of interest are considered important for comparison. The application itself has been selected to demonstrate what may be available in a variety of end-user systems.

Variation Of Number Of Processors

One of the major concerns in this experiment is estimating what can be accomplished with a Parallel PC (PPC). Although it is known that a 32 processor PPC will be available shortly, the tests presented here have been limited to a 16 processor PPC. In fact, the experiments described here have generally not exceeded 12 processors. By staying within the 32 processor framework, some of the delay factors, e.g., processor footprint described in Chapter 17, are eliminated. Variations in results when placing 3 or 4 IND modules on a processor appear sufficient to verify the conclusions about this factor.

Clocking Sample Times

The initial approach to obtaining run-time differences used the Win 7 OS to sample the real-time clock. Although the real-time clock may be very accurate, the sampling approach used by the OS appears to be on a 15.5 millisecond “heartbeat” boundary, dramatically reducing the accuracy required for these experiments. The effects of this heartbeat were to render the sampling times obviously erroneous. To rectify this situation, a VPOS *Timer* was built that runs on a separate processor, producing sample times (relative to a start time) that are accurate to within 10 nanoseconds, more than a factor of 10^6 improvement in accuracy.

Timer sampling is used many times during the course of a run to obtain multiple speed measures during different phases of the scenario. This also allows the scenario to be changed interactively as one observes the real-time outputs of the timer samples.

Capture Of Run-Time Samples

To compare speeds, a run-time time sample period must be selected that covers sufficient operations to produce a valid set of samples for run-time comparisons. For example, the run-time period selected here coincides with a complete orbit of a satellite. Using the GPS orbits, each orbit is completed in 12 hours or 720 minutes. Based upon various scenario changes, this appears to be a sufficiently long time to obtain valid statistics for each run-time sample. Figure 18-12 provides an illustration of the time samples taken from the OS clock by the heartbeat sampler, and the corresponding erroneous time differences produced.

```
12/28/13 GENERAL SIMULATION SYSTEM    GLOBAL_PLANNER
GLOBAL PLANNING TOOL

          1 SIMULATION(S) REQUIRED

START_TIME = .5027453E+000
END_TIME   = .5031946E+000
*** TOTAL_TIME = .4492968E+002 (SECS) ****

START_TIME = .5031946E+000
END_TIME   = .5037394E+000
*** TOTAL_TIME = .5448046E+002 (SECS) ****

START_TIME = .5079707E+000
END_TIME   = .5085610E+000
*** TOTAL_TIME = .5903125E+002 (SECS) ****

START_TIME = .5085610E+000
END_TIME   = .5090307E+000
*** TOTAL_TIME = .4696875E+002 (SECS) ****

START_TIME = .5090307E+000
END_TIME   = .5094800E+000
*** TOTAL_TIME = .4492968E+002 (SECS) ****

START_TIME = .5094800E+000
.
.
.
```

Figure 18-12. Erroneous heartbeat sampled clock outputs.

To solve this problem, the VPOS Timer is sampled at the start and end of the measured period, and the difference is computed to determine the time it took to complete an orbit. Since the orbits are changing, one must take multiple samples to observe differences in time results to see if particular orbits are causing significant changes.

The START_TIME and END_TIME time samples are now taken from the VPOS Timer. Since 7 decimal places are used, time samples are provided down to micro seconds. Real-time measures to hundredths of seconds are sufficiently accurate for the single orbit measure.

Capture Of Interval Samples

When using parallel processors, one must ensure that results are complete and consistent with those on a single processor. This is accomplished through synchronization of IP resources within a predetermined DELTA_T interval. The approach for establishing the size of the synchronization interval to meet the accuracy constraints is described in Chapter 14. The interval size selected for these experiments is 4 minutes producing over 180 intervals in a single orbit. This is easily changed in the Control Specification. We note that no differences between single processor and parallel processor results (other than speed) were observed in these experiments. This is due to the relatively stationary nature of the scenario.

Given the DELTA_T interval, VPOS provides facilities for capturing the Start-Time and End-Time for each IND module on each processor within each interval. These facilities generate a file for post-processing the sample data. The post processing facilities include the ability to generate graphical plots of the sample results for each processor within each interval from a specified starting interval to a specified ending interval within an overall run-time sample. These plots are shown for each test below.

Variation Of Scenarios

The scenarios tested here use ten IND modules representing 9 different platform modules and an I/O module. As described below, each platform module has multiple instances of that platform type. Heavily loaded IND modules are compared to lightly loaded modules by changing the number of platform instances within a module and the number of cross connectivity calculations that result. This causes the times to increase nonlinearly as shown below. Balanced versus unbalanced loading of different IND Modules are compared using similar changes in loading on an individual module basis. As the scenarios become more heavily loaded, one may expect the time to run a simulation to rise by a factor of 3 to 10.

From the tests described below, it is expected that single processor times for a heavily loaded scenario would be improved upon sufficiently using a 16 processor PPC so as to take less time than the lightly loaded scenario on a single processor PC. We must emphasize that the single processor simulation times described here are typically more than 2 to 10 times faster using the VisiSoft CAD approach when compared to other approaches currently available. From previous tests, it is expected that heavily loaded single processor times would be improved upon sufficiently using this CAD system so as to take less time than lightly loaded scenarios using another software environment.

Parameter Variations

Mean values and variances were computed for each time measurement using a sufficient number of samples based upon the mean and variance. The standard deviation was typically less than a small fraction of the mean value. These small deviations imply a very narrow distribution of the varied results, leading one to expect predictable outcomes with a small number of samples. This is due to the somewhat stationary nature of the IND modules and the number of platforms within each, contributing to a stable average.

GLOBAL_PLANNER_21 EXPERIMENT

Scenario Loading

Table 18-1 indicates the parameters that were considered in loading tests, and the corresponding mean times taken for a full satellite orbit for the single processor cases.

Table 18-1. Single processor speeds based upon unbalanced loading.

UNBALANCED LOADING	LIGHT	MEDIUM	LARGE
PARAMETER	Platforms	Platforms	Platforms
Satellites	24	24	24
F15s	4	8	16
F18s	4	8	12
A10s	2	2	12
SHIPs	6	17	24
Single Processor Orbit Time†	0.760	2.76	9.80
†Mean Time (Seconds)			

Because run-times in the above scenarios are somewhat small, it was decided to increase the size of the largest scenario. As indicated above, this causes the times to increase nonlinearly.

A More Heavily Loaded Scenario

Table 18-2 indicates the parameters that were used to produce a more heavily loaded scenario test, and the corresponding average time taken for a full satellite orbit for the single processor case. This loading provided a more substantial test, increasing the useful running times of the single processor tests. The simulation for this scenario is shown in Figure 18-15.

Table 18-2. Parallel processor speeds based upon balanced loading.

LARGEST LOADING	
PARAMETER	Platforms
Satellites	24
F15s	14
F16s	14
F18s	14
F22s	14
A10s	14
UAVs	16
E3As	7
SHIPs	36
TOTAL	153
Single Processor Orbit Time†	24.15
†Mean Time (Seconds)	

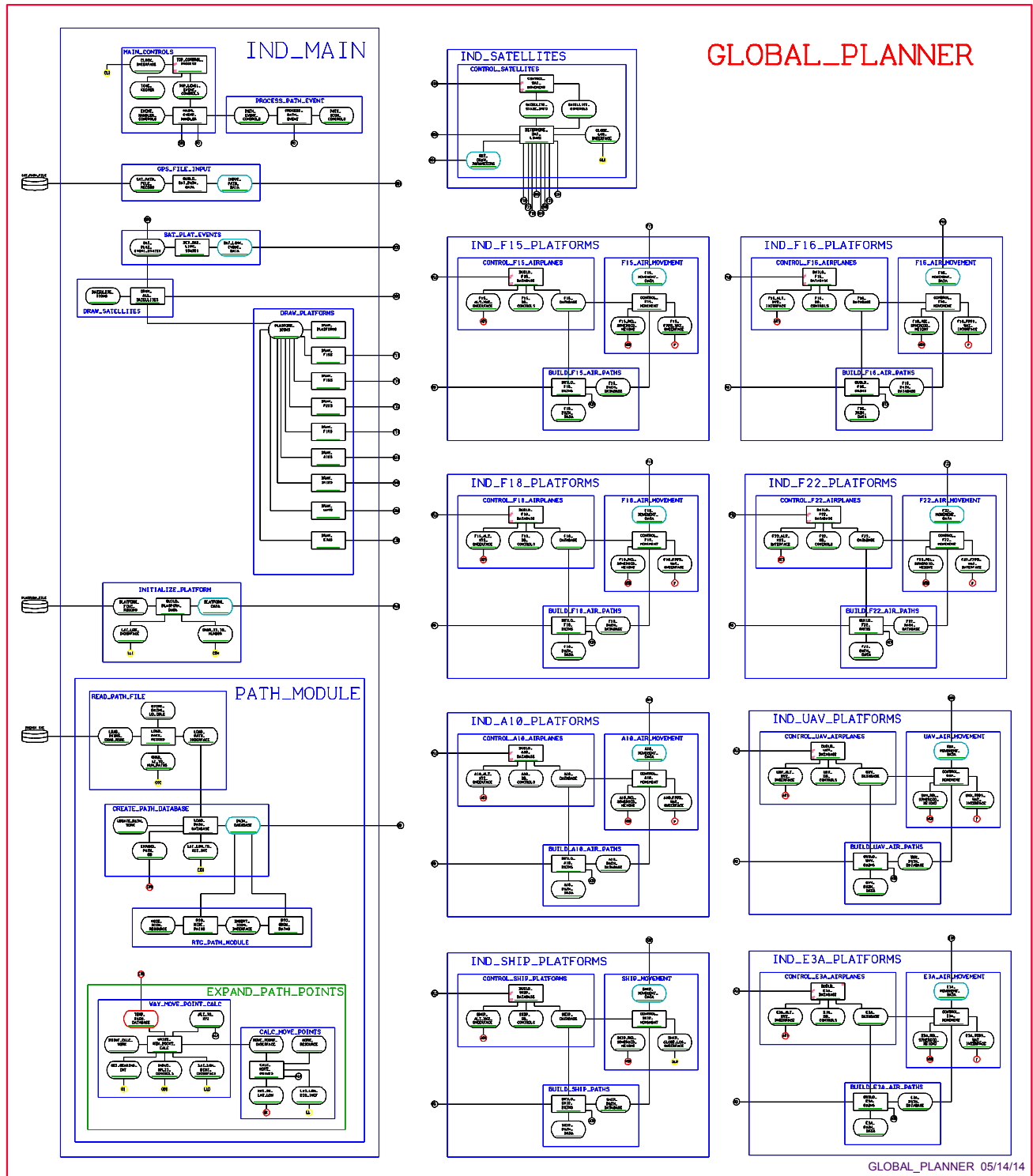


Figure 18-13. GLOBAL_PLANNER 21 - an expanded parallel processor simulation.

The scenario shown in the Table 18-2 was used in the tests described below. The expanded GLOBAL_PLANNER_21 simulation used a total of 153 platforms in 10 IND modules as shown in the above table and figure. Runs were made using a single processor, as well as using 2, 5, 7 and 10 processors in parallel. The tests described below are for the 2, 5, and 10 processor cases, followed by another 7 processor case with the UAV platforms divided into two separate IND modules. This last case shows the ability to divide IND modules to cut their times in half. This may use more processors to gain speed, or to provide a better use of existing processors.

Elimination Of Cross-Schedules During Initialization

We note that the architecture in Figure 18-11 showed Cross-Schedules used to start threads for initialization. This introduced a slight difficulty when moving IND modules to different processors, since the initiation of databases depended on start times and thus where they resided. With a minor redesign, all IND modules are now initialized independently. This independence allows modules to be moved to any processors without concern for cross-schedules. This architectural approach has worked in all cases to date, eliminating changes to relocated IND modules, another simplification provided by the property of independence.

GLOBAL_PLANNER TEST RESULTS

GLOBAL_PLANNER_21 - 2 Processor Case

GLOBAL_PLANNER_21 experiments started using 2 processors to house all of the 10 IND modules. Using 2 processors, the orbit time was about 12 seconds. Compared to a single processor time of 24.15 seconds, this produced a Speed Multiplier (SM) of about 2.0. Using the measures of useful times from the test run data portrayed in Figure 18-14, the corresponding PUE averaged more than 99%. Multiplying the lowest value of the calculated PUE times the number of processors yielded an SM of 1.99.

Figure 18-14 shows the times and charts for GLOBAL_PLANNER_21 runs with 10 IND modules on 2 processors. The Mean Orbit Time is about 12 seconds and the PUE is 99%.

GLOBAL_PLANNER_21 - 10 Processor Case

GLOBAL_PLANNER_21 testing followed with the use of 10 processors to house each of the 10 IND modules on a separate processor for “maximum” speed. This resulted in an orbit time of 4.19 seconds. Compared to a single processor time of 24.15 seconds, this produced an SM of 5.764. Using the measures of useful times from the test run data portrayed in Figure 18-15, the corresponding PUE ranged from about 60% to about 65%. Multiplying the lowest value of the calculated PUE times the number of processors yielded an SM of 6.0.

Figure 18-15 shows the times and charts for PAR_21 runs using 10 processors to house the 10 IND modules. The first module was used only for initialization and that time was insignificant, so the first processor is not shown on the chart. The Mean Orbit Time is 4.19 seconds and the PUE varies between 60% to 65%.

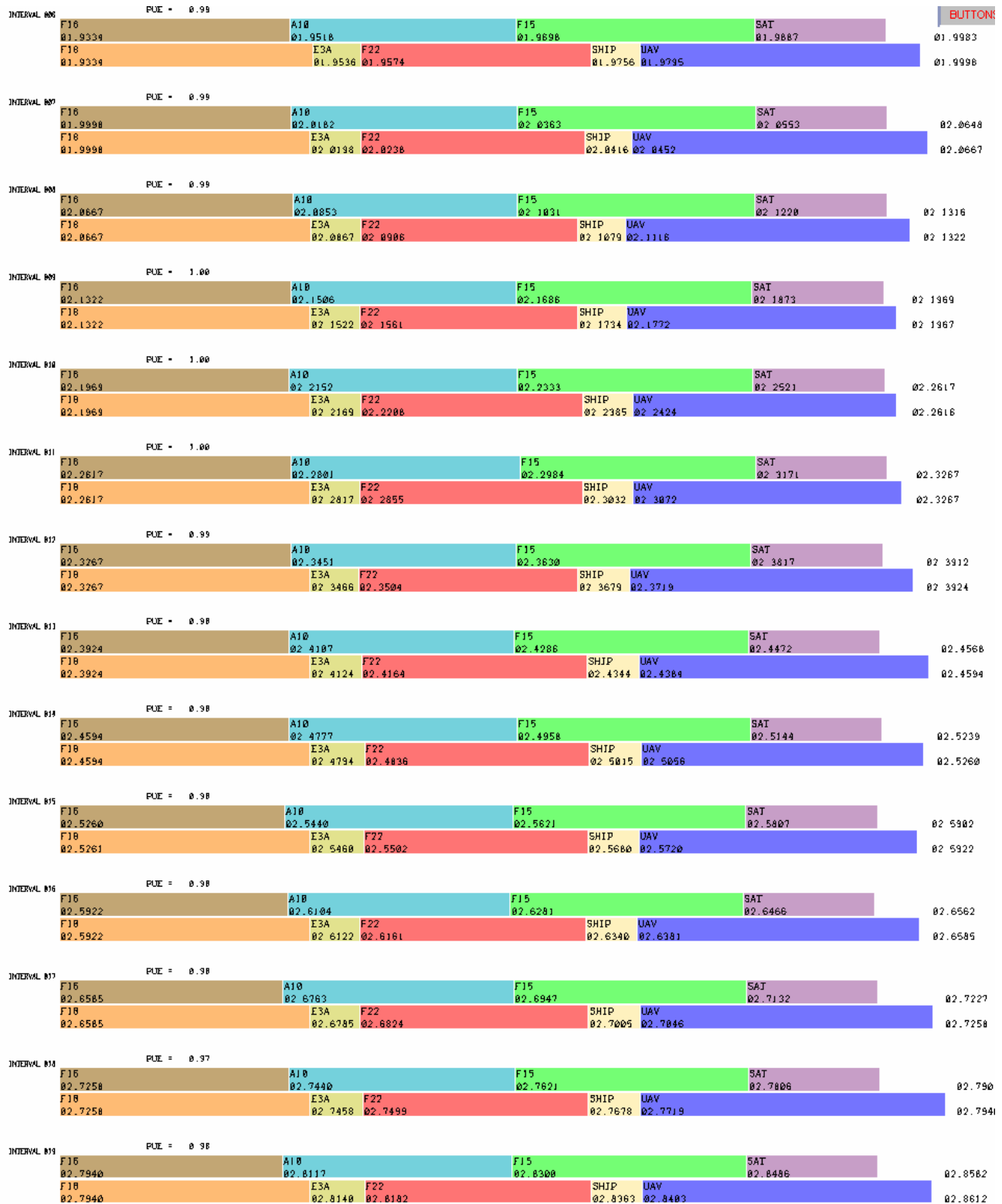


Figure 18-14. Snapshot of PUE for 2 processor case.



Figure 18-15. Snapshot of PUE for 10 processor case (9 processors shown).

GLOBAL_PLANNER_21 - 7 Processor Case

With 7 processors shown in Figure 18-16, the orbit time was also 4.19 seconds, matching the 10 processor case. This is because the UAV platform still took the longest time with the occasional exception when the F18 ran longer. Compared to a single processor time of 24.15 seconds, this produced an SM of 5.764 and corresponding PUE of about 92%.

GLOBAL_PLANNER_21 - 9 Processor Case

Now consider that the time constraint is to close to 2 seconds. This implies cutting the run time in half while running on a 16 processor PC. Looking at the charts for the 7 and 10 processor cases, it appears that the time could be cut in half by splitting each of the large IND modules into two. In addition, the SAT module may be placed on a single processor while the E3A and SHIP modules occupy a single processor with the initialization module.

Figure 18-17 illustrates the approach to dividing largest IND module into two IND modules to provide a more efficient use of processors while at the same time gaining close to a factor of 2 in speed. In this experiment, the largest IND module, the UAV, was split into two IND modules to illustrate the results. In this particular test, the original UAV module took approximately 26.0 milliseconds (msecs) to run. In the split case, the longest running time of the split modules took approximately 12.3 msecs. Based upon this result, one can expect the split modules to run in approximately half the time.

The remaining 5 large IND modules can also be split in the same manner to cut the overall run time in half. We note that, using this graphical approach, the splitting of modules can be tested easily and results plotted to determine the best fits. In addition, when there are a number of modules that are split, the statistical results of their individual variations will generally become more stationary.

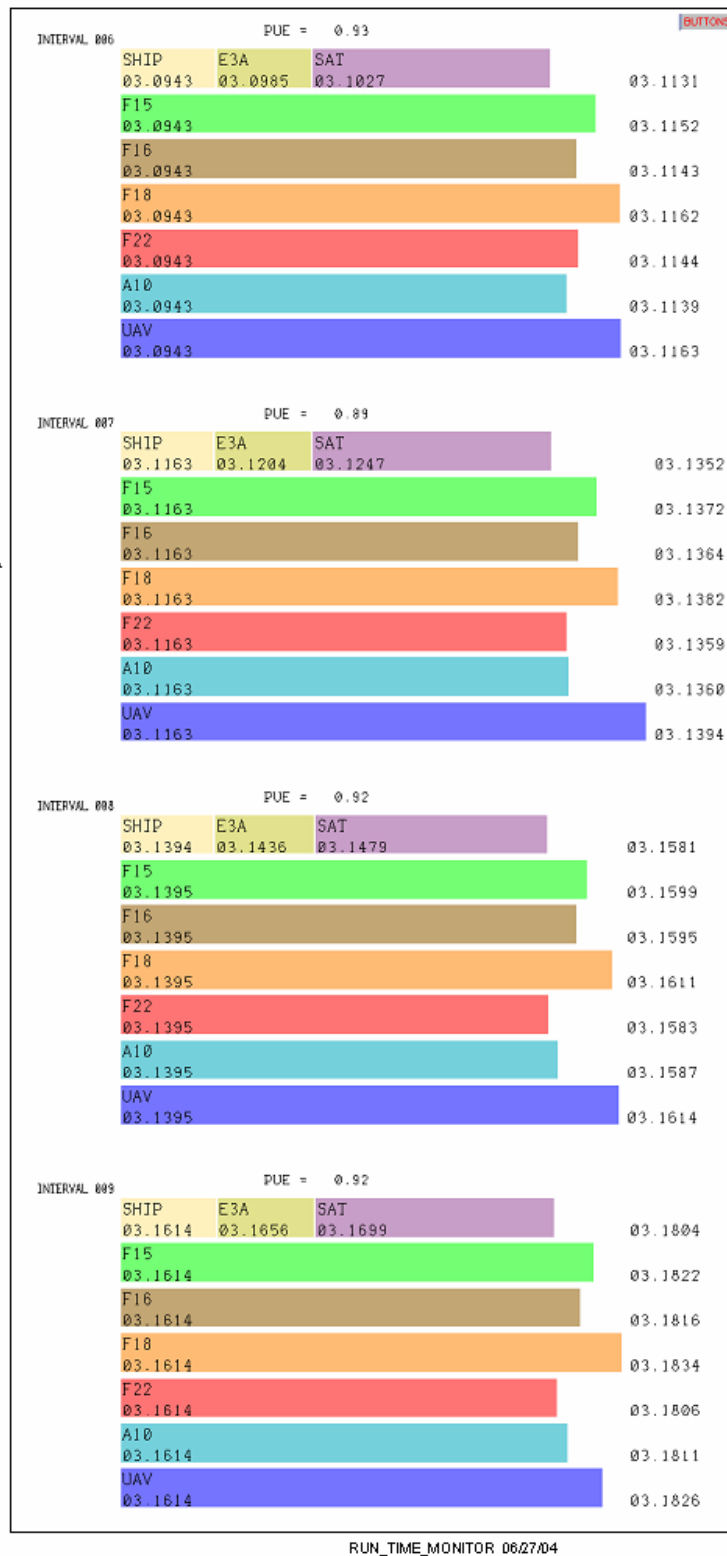
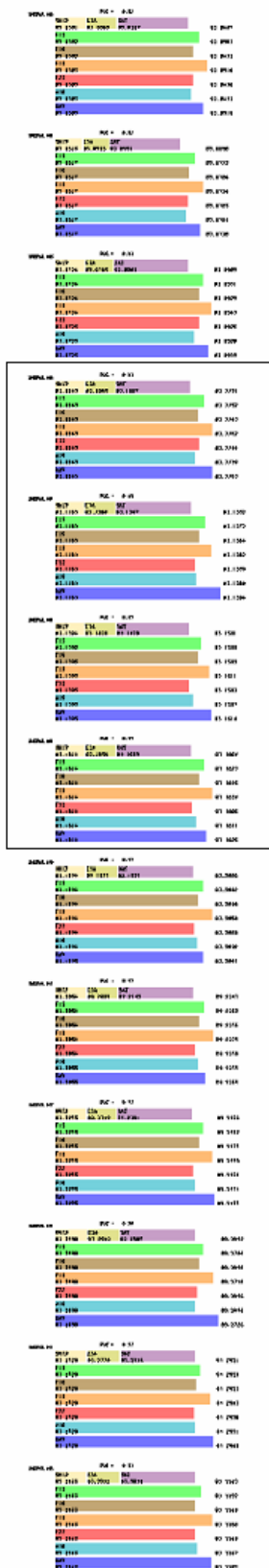


Figure 18-16. Snapshot of PUE for 7 processor case.

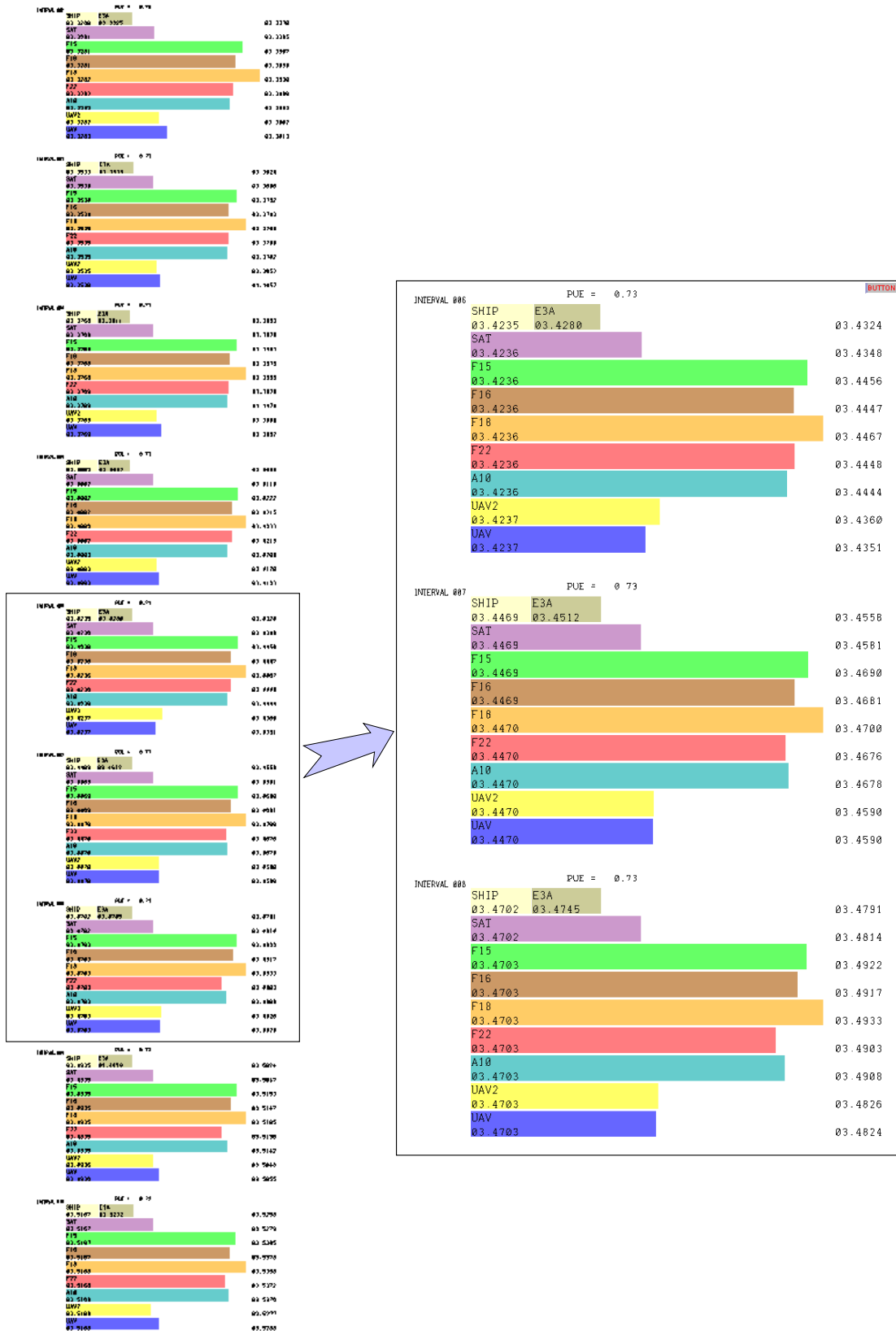


Figure 18-17. Snapshot of PUE for 9 processor case.

GLOBAL_PLANNER_28 - 14 Processor Case

To cut the orbit time of 4.19 seconds approximately in half, one must double the speed over the 7 processor case shown in Figure 18-16. Because of the high PUE achieved in the 7 processor case, this requires using at least twice as many processors. Assuming the desired constraint can be met using 14 processors, the architecture for this approach is shown in Figure 18-18. In order to split the additional 5 large running time platforms (Figure 17) onto two separate processors, a total of 16 IND modules is required. With the two small modules running on the same processor as IND_MAIN, a total of 14 processors is required. The resulting times are shown in Figure 18-19. Using this architecture, the mean orbit time was cut from 4.19 seconds to 2.31 seconds, satisfying the speed constraint. The resulting SM over the single processor time was 10.45, yielding a PUE of 75%, a reasonable result.

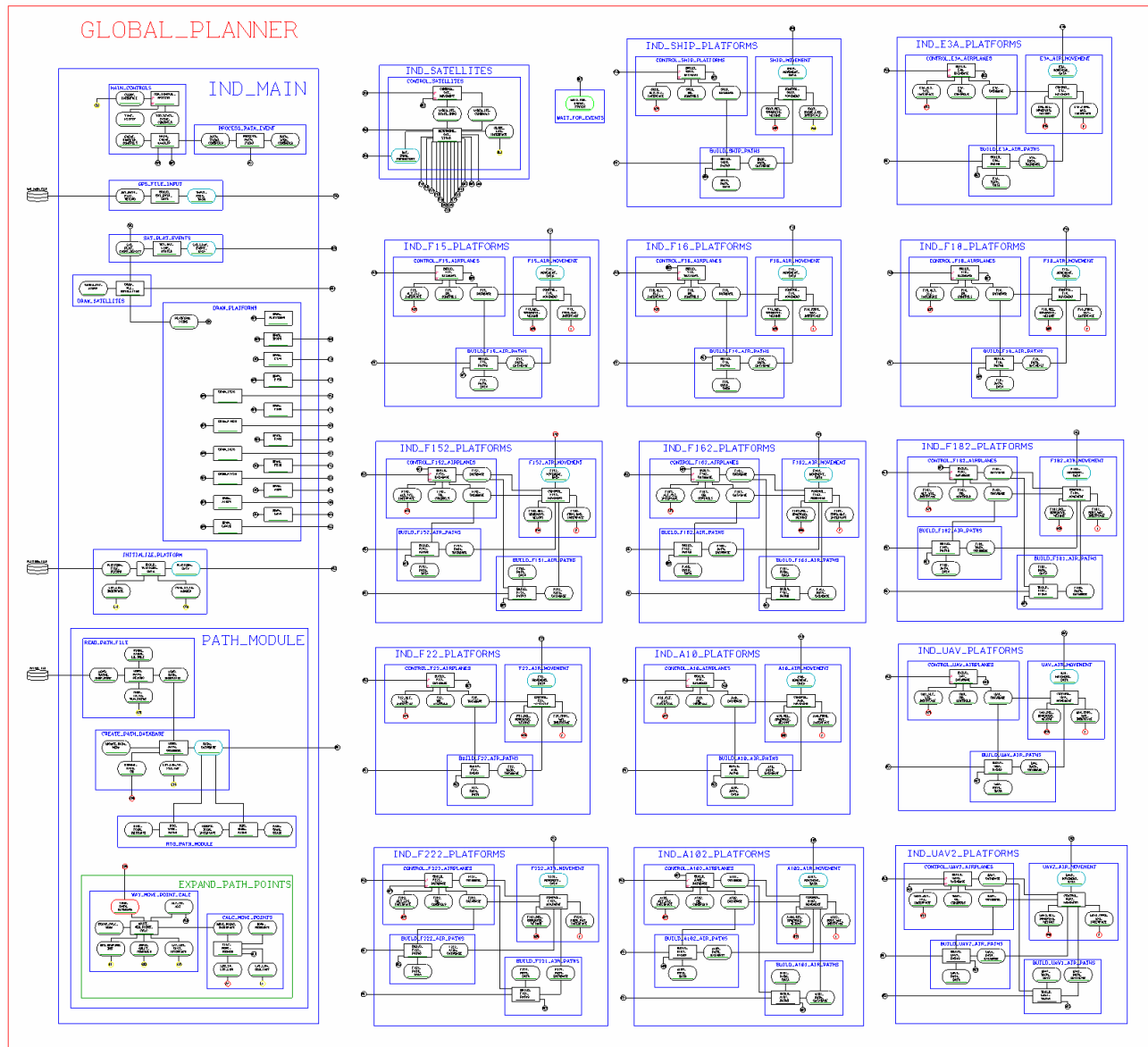


Figure 18-18. GLOBAL_PLANNER 28 - 16 IND modules running on 14 processors.

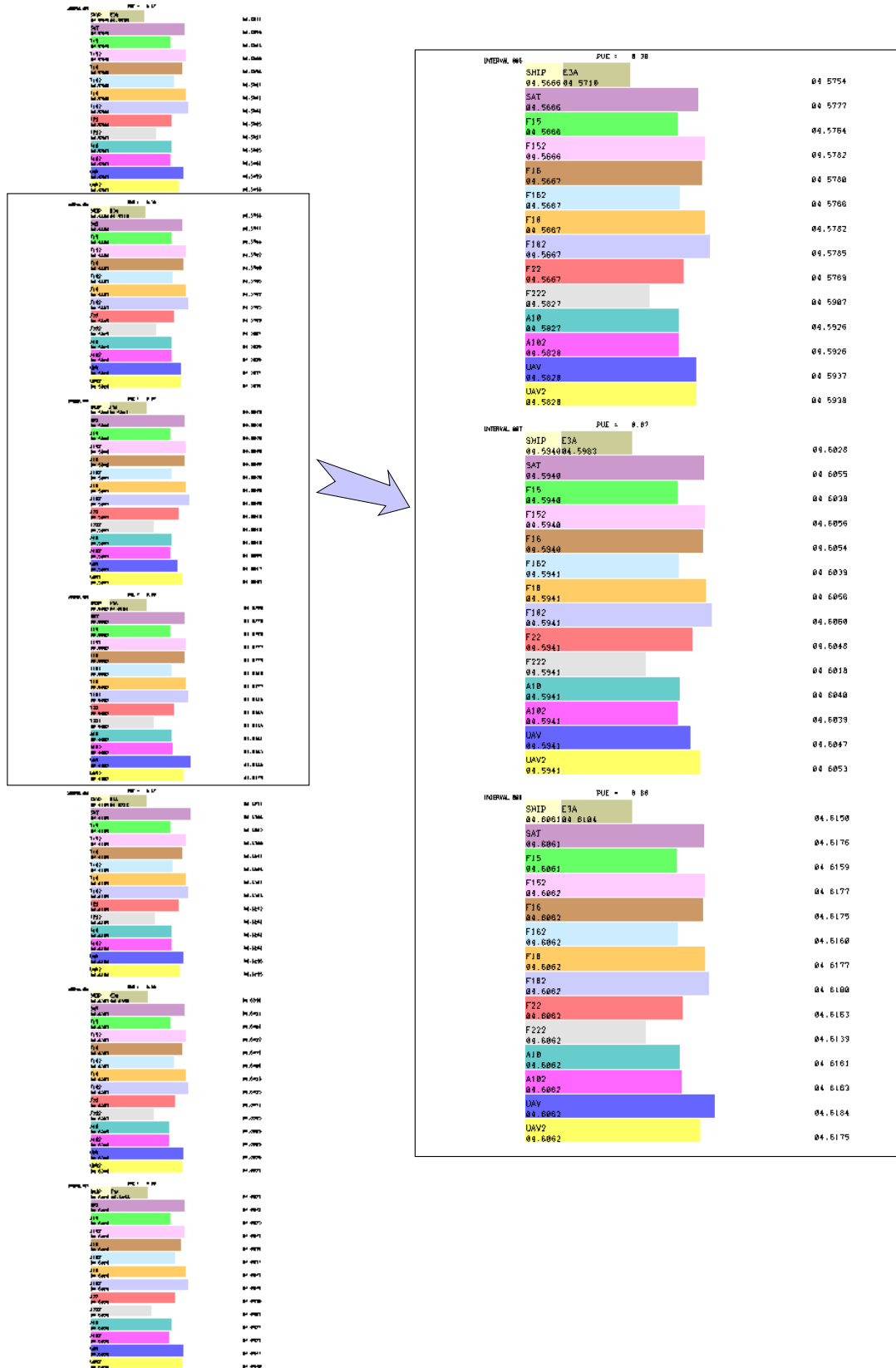


Figure 18-19. Snapshot of PUE for the 14 processor case.

OBSERVATIONS RESULTING FROM THE EXPERIMENTS

This section offers observations on some of the more critical aspects of implementation of parallel processor simulations and software using the VisiSoft CAD system. It also describes techniques that were used to ensure the validity of results. We note that simulation presents additional complexities due to the simulation clock being separate from the real-time clock. Simulations of real-time systems - as required for high-speed platforms in the military planning example used here - are quite complex.

THE PROPERTY OF INDEPENDENCE

Of the theoretical factors affecting the ability to deal with complex software, nothing is more important than the property of independence. The ability to build, test, support and expand complex systems requires that they be decomposed into independent parts. From previous chapters, two parts are independent if a change in either part does not affect the other part. For example, with a minor change in architecture, the problem of moving and grouping IND modules on different processors without code changes resulted from the independence gained when cross-schedules were removed.

When two parts are interfaced, the property of independence implies that changes to one part must not affect the behavior at the interface to the other part. This leads to the decomposition of a part into subparts that do not interface, and thus the concept of modularity. The GLOBAL_PLANNER is an example of parallel processing software that is decomposed into IND modules where the interfaces are limited to IP resources. As illustrated in the above experiments, this allows application designers to provide for worst case scenarios while balancing loads to minimize the number of processors used, as well as reduce run times.

To maintain independence of two modules that must communicate implies implementing temporally independent interfaces using synchronized simplex channels. As in any large complex communication system, this is best implemented using a single channel in each direction that the modules must communicate (a maximum of two). This implies using a single shared resource in each direction. When implementing such channels between complex modules that share a significant amount of data, the resources become large and complex. Without the ability to organize these resources into deep hierarchies, the understandability of the algorithms in the process becomes difficult to understand. Just as many difficult problems would not be solved without the language of differential equations, the underlying VisiSoft resource and process languages are the keys to achieving the independence necessary for the parallel processor solution.

MODULARITY

Decomposition of complex systems into modules is key to achieving the property of independence. With the proper decomposition, most parts can be designed to be independent so that a change in one part has a minimal, if any, effect on other parts. Complex hardware systems, e.g., computers, automobiles, airplanes, etc., must be designed to maximize reliability along with functionality while minimizing time and cost of development and support. These complex systems are excellent examples of specialized engineering approaches that result in modular decompositions that achieve these goals.

MODULE HERARCHIES

As systems become more complex, they must be decomposed further to simplify the elementary levels. If elements are too big and complex, they become difficult to design, test, support and enhance. Breaking down complexity into hierarchies of simple parts is key to overcoming these barriers. The use of hierarchies yields ease of identification, understanding and control over the details of a highly complex module.

In the case of software, it is up to the architect to decompose a system into a hierarchy of modules that takes maximum advantage of the spatial and temporal independence properties of a system. Designing the IND modules and IP resources that provide the interfaces is clearly an architectural skill that requires an understanding of the system being built.

GRAPHICAL DEPICTION OF BEHAVIOR

When testing complex software systems, visualization of system behavior is critical to discerning correct operation. Graphical depiction of the operational movements of elements of the GLOBAL_PLANNER simulation makes this point obvious. Simply watching a visual scenario unfold replaces the need to scan reams of printed output, while providing immediate recognition of faulty operations. In this environment, a picture is clearly worth a thousand words. This is another area where accurate graphical representation of a physical system is required.

In some applications, one must be creative when producing graphical images that depict operational outcomes. That is not the case with the military application provided here. The main challenge in these applications is the accuracy with which the graphics depicts the interrelated physical operations as they unfold. This facility has taken years to develop, but has paid off with the results of high prediction accuracy when planning complex operations.

DRAWING MULTIPLE CONNECTED PLATFORMS

When moving and drawing multiple platforms to evaluate connectivity, with multiple types of connect lines between them, one must synchronize the databases required to do the draws. This typically involves arriving at a point where all databases are updated so that one can proceed to do the interconnected draws. Since it is also faster to do all of the draws at once, it is best to do them at the very beginning of an interval.

PARALLEL PROCESSOR TIMING AND SYNCHRONIZATION

When scheduling processes, especially across IND modules, the actual (real-time) scheduled times are generally skewed within a DELTA_T synchronization interval. Even within a module, a schedule from outside may come in to be placed before the current (last scheduled) time on that processor, and must be brought forward to the current time within the interval. Depending upon what was previously scheduled, it may be put close to (or at) the end of the interval. Designs of interacting modules must take these possibilities into account.

In the case of simulation, it is the synchronization that must be designed to accurately represent real systems. As an example, good communication system designs ensure that variations in message arrival times will not cause a failure. This provides for distributions that represent the allowed intervals in the real system and correspond to the DELTA_T intervals used here.

When wanting to get into the next Delta_T interval, one need only add the Delta_T time to the existing time (SCHEDULE ... IN DELTA_T) --- provided that the existing time is beyond the starting boundary in the current interval by a finite amount. This facility is important during initialization, where one set of databases must exist prior to creating another set.

INITIALIZATION

When performing initialization with the current process at time = 0, incrementing by DELTA_T puts the local clock at the end of the interval boundary, not in the next time interval. Also, after scheduling in DELTA_T + ϵ to ensure getting to the next interval, one must be careful not to use this same statement successively unless one wants to move steadily ahead in the following set of intervals. From there on, one need only move by DELTA_T to get to the same spot in the next interval. This is true for single as well as parallel processor systems.

When designing modules for initialization, it helps to lay out a diagram of when different processes in different IND_MODULES are to be scheduled in terms of who must follow whom. For example, the initial schedules may be in times that are within a DELTA_T interval. After initialization, they may always be scheduled in the next DELTA_T in the future so they are placed at the same relative time point within the next interval. Since they are in the next interval, which has not yet been started, they may be placed at the time scheduled in the current interval (in the module where scheduled) plus DELTA_T, and placed at that time in the next interval (possibly in another module). This is also true for single as well as parallel processor systems.

Initial STARTS And SCHEDULEs

When the GLOBAL_PLANNER was run with the large scenario, it appeared to run OK. However, when the small and medium scenarios were run, it was determined that the schedules from the top level IND modules could be placed ahead of those that were started with the START function. Given this potential problem, the PRIORITY code was used. It became a matter of setting the priorities to ensure that the schedules were ordered properly.

To solve this problem, the higher level IND modules that invoke those at the middle level were given the lowest START priorities. Since the SCHEDULE statements implied the NOW option, and the middle level modules were started at the same time, the middle level modules were given a higher starting priority. Similarly, the bottom level modules that were scheduled by the middle level modules were given the highest priority.

This approach implies making proper use of the PRIORITY codes both for STARTs as well as SCHEDULEs as defined below.

- The NOW function is imposed implicitly when no time option is used.
- Processes scheduled with the NOW option include the PRIORITY code as well as the current time of the process doing the scheduling.
- When processes are STARTed, the corresponding SCHEDULE statements are entered into the schedule at time 0 before any cross schedules occur with the selected START priorities.
- The PRIORITY codes are invoked when placing cross SCHEDULEs in the schedule queue.

ELIMINATING CROSS-SCHEDULES

The final solution to the problem of synchronization between IND modules was to eliminate all of the cross-SCHEDULEs, including those used during initialization. This required having all schedules internal to each IND module. This, in turn, required sharing those data bases within the other IND modules that affected the internal operations of a given module. This solution was quite simple to implement, easy to understand, and reliable. Simplicity of the implementation is the result of the language, particularly the EVENT synchronization statements (i.e., the set, condition, and wait-until statements) and the IP_RESOURCE synchronization statements (i.e., the release and access statements).

SYNCHRONIZATION OF SHARED INTER-PROCESSOR DATABASES

One can start updating databases based upon when they are to be used. For example, databases can be updated so they are all ready at the beginning of the next interval and not changed while being used. This implies determining when IP resources are to be updated and when the IND modules sharing them are to be invoked to read them.

Depending upon their design, database updates could get out of sync during an interval. If they are all updated and made available at or before the end of the current interval, then they all can be read and used reliably at the beginning of the next interval. For example, one may choose a time into the start of the next interval (e.g., $0.01 * \Delta T$ from the beginning of the interval) within which IP resources can be used reliably by all IND modules without being changed. After that time, IP resources may be updated by any of the IND modules.

Again, simplicity of such implementations is the result of the language. Without the EVENT synchronization statements and the IP_RESOURCE synchronization statements, implementation of IND module synchronization would be quite difficult. More important is the requirement to understand the underlying application. Without detailed knowledge of the system being built, huge amounts of time are easily wasted. This has been particularly obvious in simulation, leading to the development of CAD systems for use by subject area experts.

COMBINING AND SPLITTING IND MODULES

As illustrated in the above experiments, it is easy to combine multiple IND modules onto a single processor, especially when there are no cross-SCHEDULES. This helps to reduce the number of processors without increasing the run time. It is also relatively easy to split large modules into separate IND modules to cut the running times of the most heavily loaded modules. Clearly this depends upon the module architecture, something that must be considered when producing the original design.

REVIEWING SPEED RESULTS

From the above snapshots of the charts produced by this CAD system, it is apparent that these facilities are designed to easily support minimization of the number of processors used to meet a speed constraint. The data is automatically collected behind the scenes with virtually no overhead. This data is then used to produce sorted files that are input to the visualization task. When this task is run, one uses the visualized output to scan and compare results. This allows many runs to be observed and compared in a matter of minutes.

CHAPTER 19

OPTIMIZING DESIGNS FOR PARALLEL PROCESSING

SUMMARY OF IMPORTANT CONSIDERATIONS

From the earliest days of computers, parallel processors have been sought to gain speed. The underlying motivation is that time is a precious resource for people who are using computers to achieve important goals. When run-times can be cut by one or more orders of magnitude, or in many cases just by whole numbers, goals are met that otherwise could not be achieved.

Today, software developers who have counted on increasing clock rates to achieve speed are being forced to use parallel processors - often against their desire. As described in Chapter 1, this has created the need for a major change in the way software is designed. Programmers can no longer sit back and watch hardware designers double their application speeds with increased clock rates every 18 months. They are also learning that current software approaches do not work in this new environment. The purpose of this chapter is to understand why parallel processor speed results vary so widely, and how to take advantage of approaches that maximize return on investment.

As defined in prior chapters, the parallel processor Speed Multiplier (SM) is a measure of the time it takes to run an application on a single processor divided by the time taken on a parallel processor. When using this measure to compare different hardware or software approaches, one must use the same frame of reference for comparison, i.e., the *fastest single processor speed* achieved for an application - independent of the hardware. Else, the speed multipliers will not fairly represent the potential economics of comparing parallel processor approaches.

Processor Utilization Efficiency (PUE) is a measure of the Speed Multiplier achieved on a parallel processor divided by the number of processors used. Alternatively, SM is equal to the product of the PUE times the number of processors used. From a software design standpoint, PUE is the most critical factor determining the speed with which an application runs on a parallel processor. Again, PUE must be calculated fairly using the same (fastest) single processor speed to compute the SM for an application.

In a real economic-oriented environment, buyers evaluate approaches based upon the cost to meet their requirements. Given that the requirements are well met by competing systems, they look to minimize their cost. Translating this to parallel processing, given that competing approaches meet the speed constraint, cost to build and operate a system becomes the major factor. This translates into facilities acquisition and operational costs as well as application system development and support costs.

When using VisiSoft, one must think in terms of minimizing the number of processors while meeting the speed constraints. This generally implies optimization of Processor Utilization Efficiency (PUE). This chapter points out the theoretical potential of the property of independence for mapping the inherent parallelism in an application system into a software architecture that achieves desired speed multipliers while minimizing the number of processors.

ANALYSIS AND COMPARISON OF FACTORS AFFECTING SPEED

Figure 19-1 illustrates the wide range of outcomes that have been produced from different software application designs when using parallel processors. Chapter 8 outlined a number of factors contributing to the speed differences illustrated in the figure. We now analyze these factors from a more advanced perspective. When faced with the design of software architectures for parallel processors, the effects of the factors described below must be understood. Of foremost importance is the goal or set of objectives one wants to achieve.

In some environments, maximizing the number of processors one can string together appears to be a major driving force. But in practical business environments, the time and cost to produce a solution weighs heavily on the approach. We are focused on those practical environments where the time and cost of both development and on-going operations presents constraints on the approach. This includes the cost to change or upgrade a system as well as the cost of its on-going operational use. The bottom line is that the factors affecting Return On Investment are critical when evaluating approaches to using parallel processors.

Looking at the top curve in Figure 19-1, one may immediately question how a parallel processor speed multiplier can be greater than 100% of the number of processors. This gets back to Bailey's paper "Twelve Ways To Fool The Masses When Giving Performance Results On Parallel Computers," [8]. If A can run 10 times faster than B on a single processor, what does A compare to? B may be running the single processor version intentionally slow to provide a very optimistic parallel processor outlook.

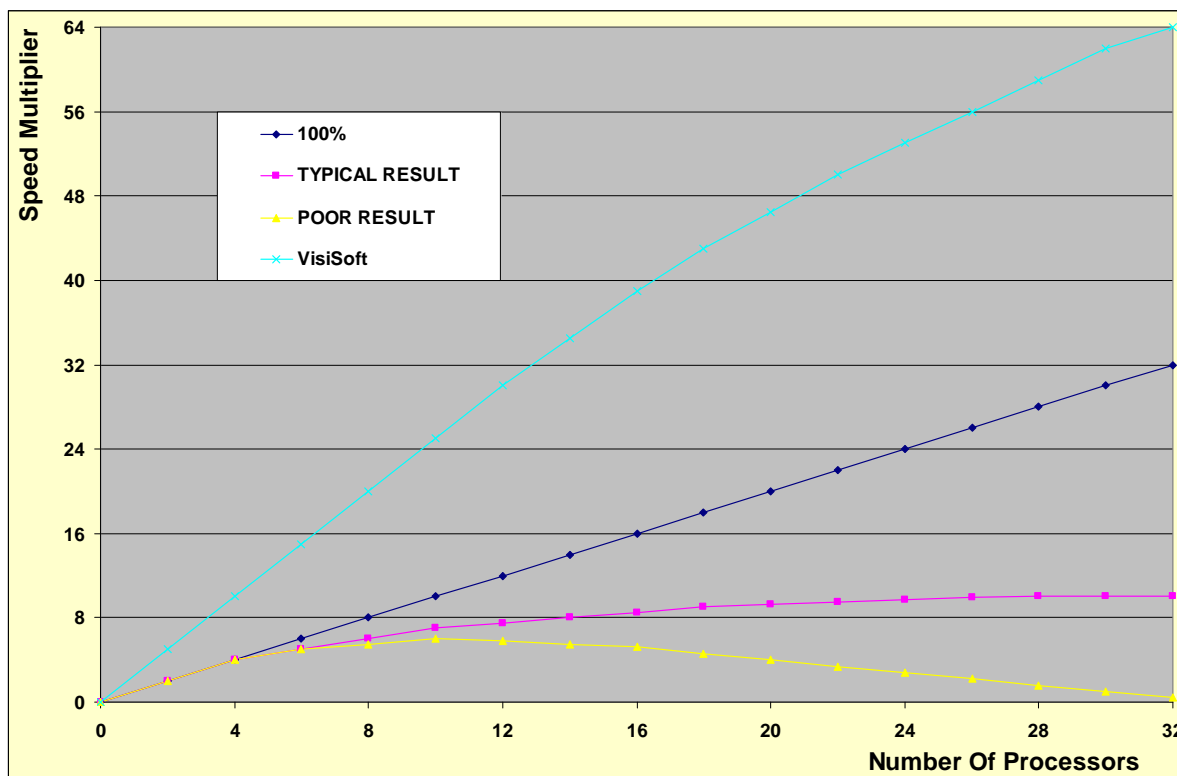


Figure 19-1. Parallel processor speed multipliers for different application designs.

Using Single Processor Speed Multipliers For Fair Comparison

Parallel processor speed multipliers may be viewed simply as the ratio of the time it takes to run an application on a single processor divided by that on a parallel processor. This implies that approaches are compared in a fair manner, not one of those described by Bailey, [8]. As shown in Chapter 17, single processor times can vary dramatically based upon the software design as well as the hardware. When reviewing two software approaches, if each is running on different hardware and one machine is clearly faster, one cannot obtain a fair comparison. Similarly, when comparing hardware designs, one must use the same software approach. Our interest here is comparing different software approaches on a fair basis.

If one software approach is obviously much faster than another - on the same single processor, then the parallel processor speed multipliers can be expected to be much faster for that approach. To produce a fair measure of their speed multiplier, one must use an accepted single processor time (e.g., one produced by the fastest approach) divided by that of one's own approach on the parallel processor. For example, speeds achieved on a single processor using VisiSoft are typically much faster than those using current "advanced" software development environments on the same machine. Based upon many comparisons, the range is from 2 to more than 10. This is a major factor when determining parallel processor speed multipliers. When compared to other approaches using the same single processor time achieved by the other approaches, it results in the blue VisiSoft curve in Figure 19-1. When compared against its own fast single processor approach, it will lie somewhere below the 100% curve.

Inherent Parallelism In The Application

Clearly the nature of the application will have a major effect on potential speed multipliers. This is because of the inherent parallelism in the application itself. An application with very little inherent parallelism will produce small speed multipliers with any approach.

Alternatively, embarrassingly parallel applications represented by totally independent tasks running concurrently should produce speed multipliers of N , where N is equal to the number of independent tasks running on N processors. But here again, when comparing different approaches, the actual running times will depend upon the single processor speeds, so each must use the fastest (or at least the same) single processor speed to produce a fair comparison of speed multipliers. As stated above, since embarrassingly parallel applications can be run effectively as separate tasks, they are simple to implement and not of interest here.

Effects Of Software Architecture

Using VisiSoft, one can design software architectures that produce an optimized mapping of the inherent parallelism of a system into independent modules while automatically ensuring their synchronization. But even then, one must compare the outcomes of different approaches.

Figure 19-2 illustrates the possible results from software designs that are all optimized for a given number of processors. By this we mean that the speed multiplier is maximized for the number of processors where, in this case, the processor count ranges from 1 to 128.

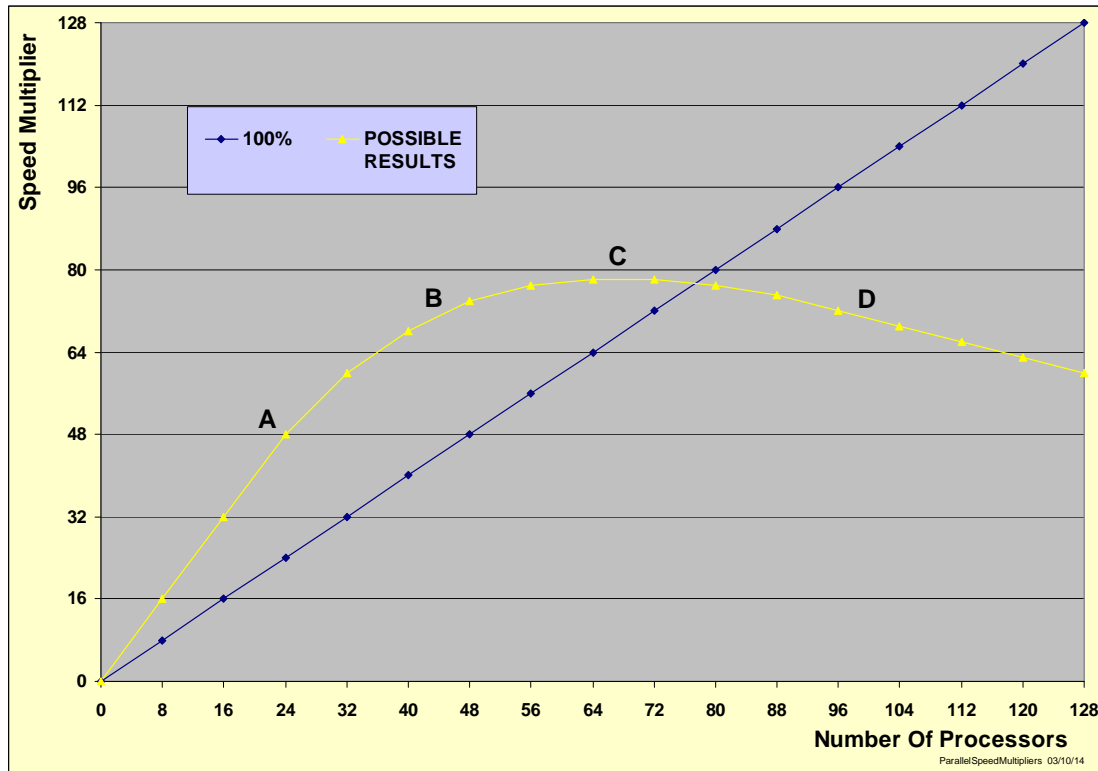


Figure 19-2. Speed multipliers as a function of number of processors.

There are multiple factors affecting the curve of results in the above figure. The first is that in region A the single processor speed multiplier is at least twice that of the other design used for comparison. In the single processor speed tests in Chapter 17 comparing VisiSoft to other languages, the single processor speed multiplier actually ranged from 1 to 2 orders of magnitude faster. The additional phenomena are best explained in terms of IND modules.

If the inherent parallelism in a system is represented by 48 IND modules, then one can expect to obtain reasonable speed gains as the number of processors is increased to 48. If the loads carried by IND modules are the same, then speed increases linearly as in the A area. When loads carried by each IND module are different, with the largest being assigned to separate processors first, increases in the speed multiplier will fall off with the load as in the B area.

As the number of processors exceeds 48, one may start to look for inherent parallelism within an IND module. Depending upon the actual loading doing useful processing, smaller increases may be gained and then peak moving into the C area. As the number of processors exceeds 72, moving into the D area, the best design is that which minimizes the fall-off of the speed multiplier. In other words, adding more processors slows things down. This is because the inherent parallelism in the system has been wrung out relative to the overhead encountered.

One may question designs that exceed the C area, but this phenomenon has been reported numerous times, e.g., in Proceedings of the Society for Computer Simulation. A point is reached where adding more processors actually reduces speed, requiring more time to complete a run. The critical result is that one must optimize the design based upon the inherent parallelism in the system, accounting for constraints on time and cost for development and operations.

The C area is further exacerbated by systems requiring large numbers of processors, where the memory boundary crossing delays take a significant toll on the speed multipliers. When a system is spread across multiple trays in a large parallel processor, communications between trays becomes a significant delay factor. When faced with multiple racks of computers, this problem is much further compounded. Such systems are most effective when applied to very special applications, ones that are typically embarrassingly parallel.

Designing For Worst-Case Application Scenarios

Given that the software has been well designed for a given application, one may be faced with different scenarios. This is particularly true in transaction processing systems containing a back-end database, where the number of transactions per minute will depend upon the time of day. This effect is also true in simulation. Chapter 9 showed the architecture for the global planner simulation, where large numbers of satellites communicate with large numbers of airborne platforms. This is expanded in Chapter 18. In these simulations, the scenario can have a major impact on the speed multiplier, because it determines the amount of loading on each of the IND modules. With good IND module architectures, the multipliers go up as the modules are more heavily loaded.

Additionally, the multiplier grows higher as the loads are balanced. This is because modules with light loads will be waiting on idle processors for those with heavy loads. Load balancing becomes a design trade-off. Typically one must design for worst-case conditions. For example, if the loads can be balanced across IND modules, then a balanced design can provide optimal support for worst-case conditions. If the modules are inherently unbalanced, then one may consider placing multiple IND modules on a single processor to balance the load and minimize the number of processors used.

MEASURING PROCESSOR UTILIZATION EFFICIENCY

As described above, and particularly in Chapter 6, Processor Utilization Efficiency (PUE) determines the return on investment obtained when adding more processors. This is estimated to range from 40% to 95% when using an optimized number of processors. This number must be compared to the typical 10% obtained when trying to use a large numbers of processors with the hope of increasing speed.

VisiSoft IND modules are generally large and remain on a specified processor, minimizing if not eliminating swapping and paging and therefore increasing processor utilization efficiency. However, as processor loads become unbalanced, processor utilization efficiency will fall and one must carefully consider the application level design constraints and function to be optimized.

This problem is illustrated in Figure 19-3 where processor loading is shown in the green area with unused processor time shown in pink and blue. Figure 19-4 illustrates a reduction in time (about half) using an improved architecture. By grouping lightly loaded modules, the number of processors used, M, may also be less than N in Figure 19-3. We note that VisiSoft provides direct measurements of the loading as shown in Figures 18-16 and 18-19. These are from the experiments supporting the balancing theory illustrated in Figures 19-3 and 19-4.

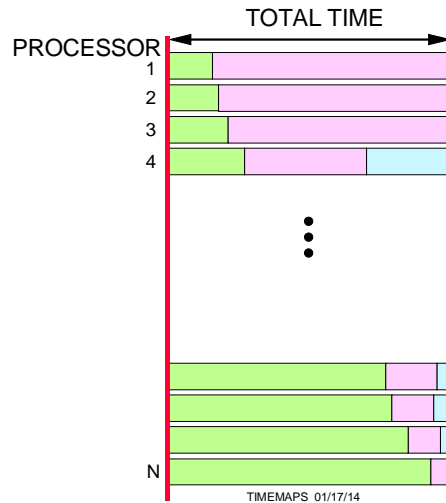


Figure 19-3. Unbalanced processor loading.

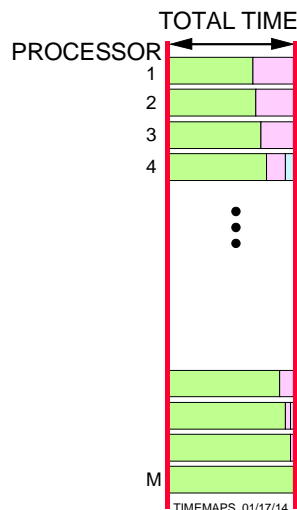


Figure 19-4. A more balanced processor loading.

ACHIEVING OPTIMAL RESOURCE SOLUTIONS

Looking at the above figures, design of parallel processor software becomes a typical nonlinear programming problem where one may vary parameters and assess the results to determine what is best for a given set of requirements for a given application. These criteria must include the cost of resources used (number of processors, electric power, air conditioning, floor space, etc.) as well as time to complete a run. They must be mapped into an optimization criteria, and a set of constraints.

In typical applications, the run time constraint may be most important, i.e., it is a hard constraint that must be met or the application will be considered a losing proposition. When the problem is posed with a run-time constraint, then one looks to minimize cost to meet the constraint. This typically involves minimizing the number of processors while ensuring the time constraint is met.

Alternatively, one may switch these criteria, making the number of processors available the hard constraint and minimizing the run time. When concerned with economics, the solution will likely be quite different. We will address the economic problem.

Meeting Run-Time Constraints While Minimizing Processor Count

When looking at Figure 19-3, one must consider the unbalanced loading from two standpoints.

- Can modules on the heavily loaded processors be split to run on separate processors?
- Can modules on the lightly loaded processors be grouped to run on a single processor?

In the example illustrated in Figure 19-4, the architecture of the IND modules on the heavily loaded processors were redesigned to take advantage of their inherent parallelism. It is important to note that, if sufficient inherent parallelism does not exist, this may not be possible.

As noted above, the number of processors (M) in Figure 19-4 may be less than that (N) used in Figure 19-3 - while cutting the run time in half. This can occur if the number of lightly loaded modules is large compared to the heavily loaded modules, and they can be grouped architecturally without changing the results. Clearly their grouping is also restricted by the time constraint.

IND Module Architecture - The Critical Component

Achieving speed constraints while minimizing the number of processors clearly depends upon the architecture of the IND modules. Their design must take maximum advantage of the inherent parallelism of the system. This is akin to hardware design wherein module independence is key. Simulations of physical systems are best developed around the design or physical properties of the system itself, since that usually maps into the best representation of inherent parallelism as well as independence. IND module architectures clearly depend upon the design of the data space, and of course the separation principle which provides for modularity itself.

Synchronization of IND Modules

As stated in Chapter 18, simplicity of IND module implementation is the result of the Resource, Process, and Control Specification languages, and particularly the ability to create large complex IP resources. In addition, without the EVENT synchronization statements and the IP_RESOURCE synchronization statements, implementation of IND module synchronization would be quite difficult. Finally the ability to quickly change assignments of IND modules to processors makes testing and reorganization simple, leading to fast minimization of the processors required to meet the application speed constraints.

OTHER FACTORS AFFECTING SPEED

The Effect Of Operating Systems On Parallel Processing

As indicated in prior chapters, this is the difference between a Windows OS, Linux OS, or a specially designed OS that attempts to allocate processors automatically - versus VPOS. Depending on the application, OS design and hardware design, a parallel processor application running under VPOS may be 2 to 10 times faster than one running under Linux or Windows. This is because the important information about the parallelism designed into the software architecture is passed on to the tailored Run-Time System (RTS). The RTS interfaces with VPOS in a manner that makes maximum use of the information to allocate physical processors at run time.

Better Use Of Chip Space

VisiSoft IND module design eliminates the use of threads across IND modules. Communication between IND modules on separate processors uses the run-time IP Communications (IPC) manager eliminating concerns for synchronization. Sharing memory with a server eliminates the need for DMA channel interfaces to external devices. Stack facilities and complex instruction caching are also eliminated. All of these architectural improvements serve to simplify parallel processor chip design allowing for more memory close to the processors, further reducing swapping and paging.

Distance Factor

All of the above factors serve to increase the speed of a system using less processors. If the number of processors is cut by a factor of 8, one may discover additional speed-up factors of 2 or more just due to a reduction in time delays caused by the distances between processors and memory. These time delays increase nonlinearly as the footprint of a system becomes larger. As shown in Chapter 6, one can expect the speed of a 32 processor PC to match that of a 250 processor HPC. Such high speed multipliers are helped by the reduced distance between processors and memory. Coupled with the other factors, they can yield comparative speed multipliers with a substantial reduction in number of processors.

SOME SAMPLE APPLICATIONS

The examples in experiments described in the prior chapter were simulations. This was done because synchronization with the simulation clock is critical to obtaining accurate prediction of outcomes of the systems being analyzed. In these cases, synchronization within the DELTA-T interval was necessary to accurately represent the unfolding of events in different IND modules running on separate processors.

Software applications must follow similar design requirements, and are likely to be easier to produce when following the approach described here. Some of these concepts are apparent from the applications described below.

The GLOBAL_PLANNER simulation shown in Figure 19-5 illustrates this concept. The green lines in the figure define direct CALLs between processes within an IND module. The red lines define Cross-SCHEDULEs by processes in one IND module to those in another.



The Cross-SCHEDULEs in the GLOBAL_PLANNER are only used for initialization. This is because the architecture is designed to limit I/O device interfaces to a separate IND module so that I/O interrupts and device handling does not occur on the set of tightly coupled parallel processors. A general form of parallel processor architecture that supports this type of IND module is illustrated in Figure 19-6. This architecture supports removal of the General Purpose OS (GPOS) time - spent managing devices - from the parallel processor domain managed by VisiSoft Parallel OS (VPOS). As a result, IND_MAIN is placed on a separate server processor that puts and gets information on the I/O devices, while sharing memory directly with the parallel processors.

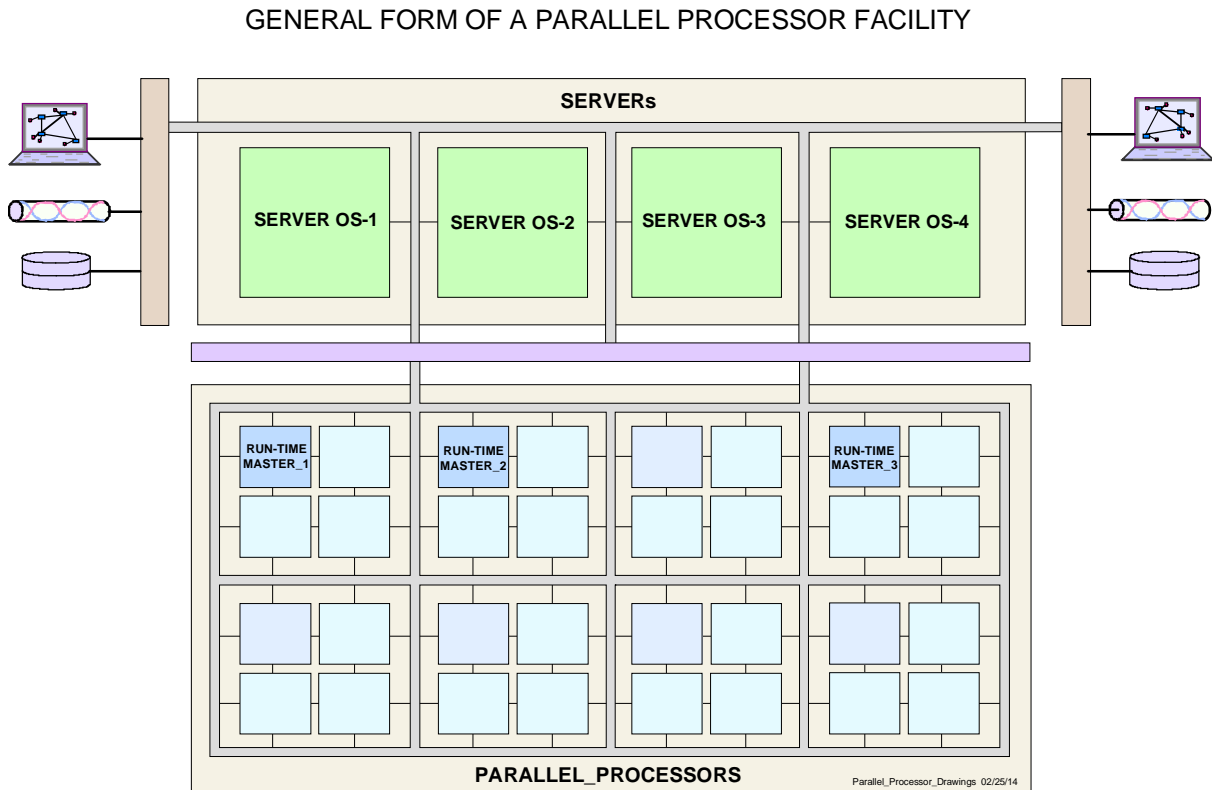


Figure 19-6. Tightly coupled parallel processors sharing memory with servers.

Once the other IND modules are initialized, they run independently. We note that they are started independently by START codes in the architecture. This implies that there are no instructions in an IND module that affect another IND module. This begs the question: Can this be done for all architectures, including those handling the I/O devices, without affecting synchronization?

Architectural Trade-Offs - A Statistical Measure Of PUE

Referring to the measures of Processor Utilization Efficiency (PUE) in Chapter 6, and particularly Figure 6-3, that figure illustrated the distribution of Processor Utilization (PU) for each processor where the green area represents useful processor time. Flipping the axes in Figure 6-3 so that the X axis is now vertical in Figure 19-7 below, and the processor number, N , is now along the new X axis, we can make a comparison of two such plots.

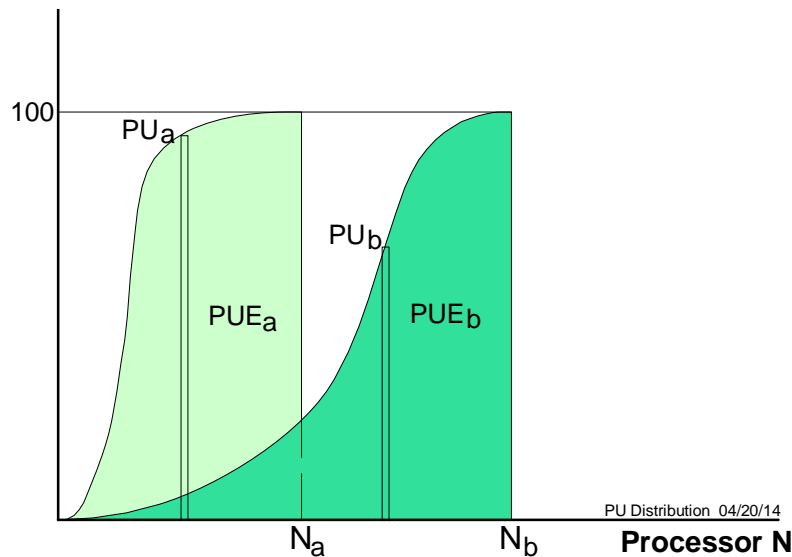


Figure 19-7. Comparison of Processor Utilization Efficiencies (Areas under the curves).

Looking at Figure 19-7, PU_a and PU_b represent the percent processor utilization for a particular processor, N . The areas under the two curves represent the PUE for two software architectures, **a** and **b**, and the number of processors each uses to meet the time constraint. PUE_a is equal to the area under the light green curve divided by the rectangular area defined by last processor number N_a . PUE_b is measured similarly, using N_b processors. Clearly PUE_b is well under 50% whereas PUE_a is greater than 50%. From the figure, architecture **a** uses almost half the number of processors, N_a , as architecture **b**, N_b , to achieve the same speed.

For this comparison we assume that the amount of useful time spent on each processor (height of the vertical strips, PU , for each processor), produces the equivalent amount of work within the same time frame on each processor. Thus, if the area under the curve for each configuration (PUE_a and PUE_b) is the same, then the amount of time taken to complete a run is the same. The validity of this assumption depends upon a number of factors, e.g., the placement of IND modules relative to the memory they access. With sufficient memory next to each processor, or if the average access time over the run is equivalent, the assumption will be valid. With IND modules that perform a fair amount of processing each relative to the ΔT window, and a well designed OS for parallel processors, this assumption should remain valid.

Producing Optimal Software Architectures

In the typical case, run times must fall within a specified time constraint to meet simulation requirements. One then looks to minimize the number of processors required to meet that constraint. We start by assuming that **b** meets the time constraint. Looking at Figure 19-7, from a statistical standpoint the variance of PU's in **a** (vertical bars PUa) is much more narrow than the variance of those in **b**. If one can design an architecture such that the variance of the PU's in **a** is minimized, it will be an optimal architecture using the least number of processors to produce the required speed. If this is done for the worst case scenario for a particular application, then the optimal software architecture will have been produced for that application.

In summary, there are two dimensions of the problem. The first is time and the second is the number of processors. One must first meet the time constraint. This may require splitting IND modules that can run in parallel. Once an architecture is produced that meets the time constraint, one looks to group IND modules onto fewer processors.

The Effect Of Independent Control On Speed

As indicated above, all of the IND modules in the GLOBAL_PLANNER simulation are independent with respect to control once initialization is complete. Even initialization may be done using independent control by scheduling IP resource checks based upon the simulation clock. This approach also holds for software systems using the same clock, which for real-time systems may be tied to the real-time clock. This begs the question: When is it more effective to use cross-schedules instead of maintaining independent control?

To address this issue, consider that the GLOBAL_PLANNER simulation includes message transmissions between platforms (and therefore IND modules) and that message receipt requires propagation path loss checks to determine if enough power is available at the receiver antenna. The fastest way to process these checks is to perform a sequence of operations. The sequence depends upon transmitters and receivers having access to the record of latest changes that affect each other.

Information Exchanges Between XMTRs and RCVRs

Figure 19-8 below illustrates the nature of time slotted communications between mobile radios, each with transmitters (XMTR) and receivers (RCVR), exchanging messages. The time slotted approach is used to increase the probability of reception in a noisy environment. Time slots (TSN, TSN+1) may be assigned in advance to support a channel between a transmitter and one or more receivers, and may be reused by other channels when free. When broadcasting messages to many receivers, the transmitters may not care if all messages were received. When messages must be received, the transmitters must receive an acknowledgement of reception.

When building a simulation of such a system, one must model the message traffic in detail to determine if and when messages are lost. This requires that the receiver models must determine if they have received sufficient power at their antennas to overcome the noise and receiver thresholds to correctly receive the desired signal. This requires determining the loss of power along the path from the transmitter to receiver, including the effects of terrain and foliage.

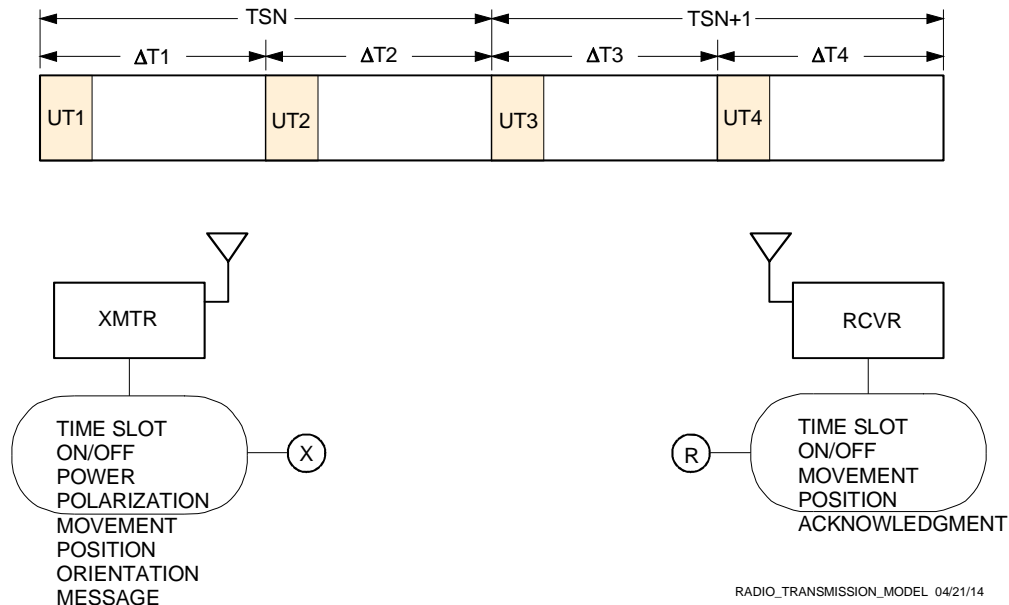


Figure 19-8. Communication protocol between transmitter and receiver.

In the actual system, message reception is determined by the electronic circuits in the receiver. In the model, the receiver must determine how much signal and noise power have hit the antenna before it can perform the computations necessary to decide on message reception within a particular time slot. For accurate models, this requires a knowledge of the information listed over the transmitter (X) resource symbol in the drawing for all transmitters transmitting in that time slot. By knowing which transmitters may send messages to that receiver in a given time slot, it can access the information it needs from the IP resources attached to those transmitters and proceed to perform the determination.

When a transmitter turns on, off, changes power, moves, or sends a message, it puts this information into the IP resource available to all receivers that receive messages from that transmitter. This is done during UT1 in the first interval, $\Delta T1$, within time slot TSN. This information is available for use by the receivers to perform the computation that determines whether the signal was received during UT2 in the second interval, $\Delta T2$, within time slot TSN. If the message requires an acknowledgment, the return message is placed in the IP resource, R, shared by that receiver with transmitters that require the information. Transmitters looking for an acknowledgment use UT3 in $\Delta T3$, within time slot TSN+1.

Alternative Solutions

The important point to be derived from this example is that there are many ways to exchange the information described above. For example, it may be done with or without Cross-SCHEDULES from transmitters and receivers. They have not been used in the approach described here. If used when performing broadcast to many receivers, scheduling may take significant additional time when confined to a single processor. However, by using cross-schedules, one may be able to support the information exchange using a single ΔT interval within a time slot TS. Because of the nature of such exchanges, this approach is subject to inaccuracies.

There are also different ways to split the work over many processors. For example, one may put all platforms of a given type on a single processor, especially if they interchange messages much more frequently than those between different platform types. This will also depend upon the computation time for each platform, and the consideration of the terrain databases required to support a single platform.

When sending messages between platforms, one must be concerned about the timing of the messages being sent and received from different platforms on the same or different processors. Specifically, messages sent in a given time slot must be received in the time slot. The following cases must be considered.

Case 1: A message is sent from a platform on one processor to a platform on another processor.

- The sending platform may cross-schedule the receiving platform to receive the message. This ensures that the message will be received within the same ΔT interval.
- The receiving platform may schedule itself to run again within the same ΔT interval. This implies that, when it runs again, the information shared in the IP resource with the sending platform has been updated with the message. This gets more complex if platforms on other processors can send messages to the receiving platform within the same time slot.

Case 2: Two platforms are grouped onto the same processor to increase the PUE

- The sending platform may cross-schedule the receiving platform to receive the message at a slightly later time. This ensures that the message will be received within the same ΔT interval.
- The receiving platform may schedule itself to run again - at a later time - within the same ΔT interval to check for messages. This implies that, when it runs again, the information shared in the resource with the sending platform has been updated with the message. This gets more complex if platforms on other processors can send messages to the receiving platform within the same time slot.

As indicated in prior sections, it is best to split the work to take advantage of multiple processors operating in parallel provided the inherent parallelism exists. However, as indicated in the fine grain model analysis, this can work against the speed constraint if the models that must exchange information run faster when combined on a single processor.

Potential approaches and decisions such as this require detailed knowledge of the application, i.e., a subject area expert. That person is in the best position to consider all of the requirements and options regarding the models in the simulation and its resulting accuracy. Improvements in speed of the simulation are determined by the placement of IND modules on processors, and the resulting data taken from runs. This data must be easily interpreted with respect to the even distribution of useful time over the available processors. One must not forget the ability to easily understand the models for purposes of validation, as well as for expansion by a newcomer to the project. Model architecture is critical to both of these important and potentially costly functions.

TRANSACTION PROCESSING EXAMPLE

Figure 19-9 illustrates a simplified look at parallel processing architectures for a transaction processing system. The starting assumption is that communication software is used to field transactions coming in from large numbers of sources over a large geographical area using multiple channels. These transactions are presented to the transaction processing subsystem in multiples stacks of transaction records, potentially in some predefined order. Transactions are to be applied against multiple databases or segments thereof depending upon parameters in the incoming record fields. Various databases may then be updated, and responses sent back to the originators as well as to other destinations via communications channels.

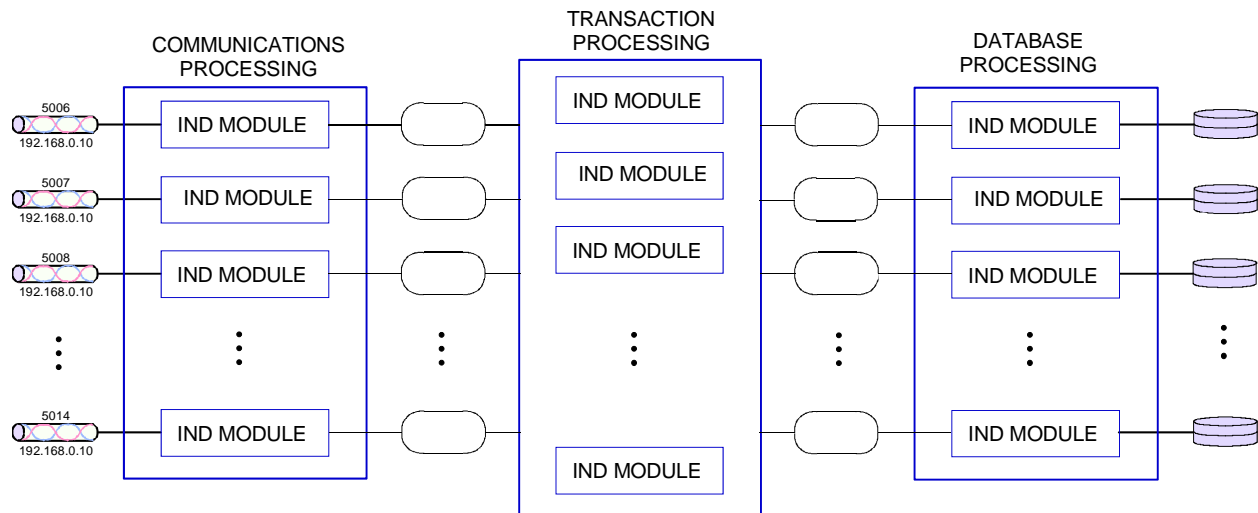


Figure 19-9. On-Line Transaction Processing.

In this particular example, IND modules contained in a communications processing subsystem are assigned to communications channels, each handling multiple incoming transactions. These modules may perform preliminary edits on the incoming transactions and stack transaction records for input to the transaction processing subsystem. IND modules within the transaction processing subsystem are assigned transactions that are further edited. Based upon one or more keys, information is then passed to and sought from the database processing subsystem where IND modules handle segments of one or more databases.

Using the architecture in this example, multiple IND modules in each subsystem operate in parallel to speed transaction processing. These modules may be instantiated within a subsystem. We note that there are many ways to control the selection of IND modules within the transaction processing subsystem and the database processing subsystem. This may be done internally where IND modules pick the next transaction to be processed. Or IND modules may be assigned by a manager of the subsystem. Selection of the number of incoming communications channels and segmentation of the databases may be changed on an irregular basis depending upon the transaction statistics.

Clearly an architecture must be tailored to the particular application. In complex systems it may be highly beneficial to simulate the architectural design of the subsystems using potential worst case incoming transaction scenarios before building or modifying the actual system.

FAST SORT-MERGE

When sorting huge files, one is faced with the optimal use of memory and particularly the corresponding trade-offs one can take advantage of when using memory to gain speed. Solutions to this problem go back to the days of punched cards and magnetic tape, and the size of core memory. This became a trade-off between sorting and merging. Core sorts were fast, but memory was limited. With file sizes that were N times larger than that which would fit into core, one had to break the files into N subfiles to perform the sorts. Subfiles were then merged using large secondary memory, e.g., magnetic tape or disk. We note that the fastest core sorts use a statistical approach to produce a sorted set of keys.

With disk, merge operations could be performed fast using blocked records tuned to the disk segment sizes. In a multi-tasking OS environment, multiple merges could be performed in parallel using different tapes or disks. A good SORT-MERGE design can select the optimal parameters based upon the layout of the records on the file, and control use of the hardware at hand to minimize the time to sort a huge file.

On a parallel processor, multiple sorts and merges can be performed in parallel. Sizes of subfiles can be tuned to the file and record layouts, the keys to be used for sorting, the number of processors, etc. Determination of the optimal parameters can be automated using a generalized model of multiple keyed files. RTS code can be automatically generated, translated and linked into a final set of IND modules that will minimize the time required to perform sort-merge operations on large files.

SEARCHING FOR CORRELATIONS IN HUGE DATABASES

To gather intelligence on various subjects, one may need to search huge databases to determine if correlation exists between certain specified properties and the contents of the database. When using relatively simple techniques, e.g., linear correlation, this may be done using embarrassingly parallel approaches. However, when the properties are non-linear or stochastic (time-varying), the property of inherent independence is lost, and one cannot expect linear correlation to work. This implies that the problem is no longer embarrassingly parallel.

Nonlinear correlation approaches are required when critical properties of the data are interdependent. This implies that information stored in one area affects the interpretation of that stored in another. This loss of independence implies that correlation searches must exchange information as the search proceeds. Such algorithms are similar to those used for nonlinear optimization described below, and are easily developed using the VisiSoft CAD approach

NONLINEAR OPTIMIZATION

Nonlinear optimization is used in engineering to produce difficult designs. An example is determining the parameters to be used in systems or devices that operate in a nonlinear fashion. By nonlinear we imply that there is no correlation between the directional derivative of design vector and the optimization function. To further understand this, consider trying to find the highest hilltop in an area with many hills. The directional derivative will find the nearest - not the highest - hill.

Similar to the fastest core sorts, solving this problem is best done using a statistical approach to locate the optimal solution. This involves taking samples to form a distribution. In complex nonlinear systems, this implies running a simulation to obtain a sample. Each simulation is generally an independent run with a different parameter vector so that each sample can be obtained from a separate parallel processor. Processing the samples to obtain a new distribution is very simple and fast compared to the nonlinear simulation itself.

REAL-TIME CONTROL SYSTEMS

Our interest here is in large complex real-time control systems. Figure 19-10 is a simple illustration of such a system. In the problems of interest, the control system may contain a large number of observable inputs provided by many sources. In certain cases, these inputs may be processed by human intelligence to help make decisions on controlling the system. Once a plan (a sequence of control inputs) is made, the corresponding control actions are promulgated down to the subordinate people or systems to be carried out.

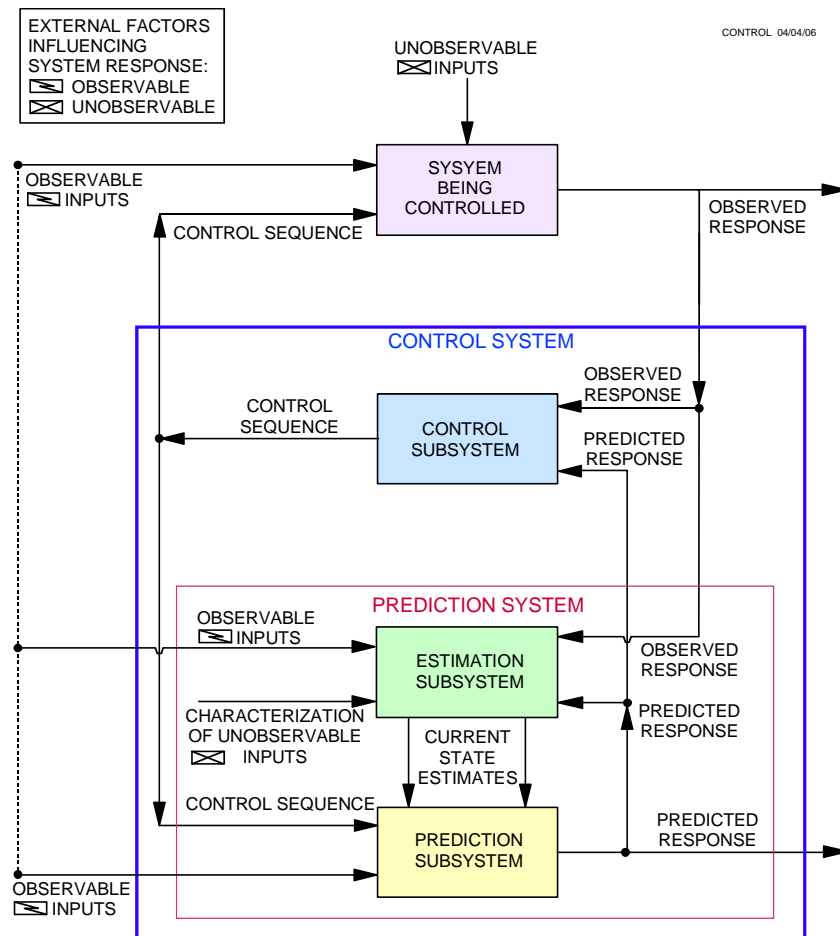


Figure 19-10. Simplified representation of a control system.

The sophisticated part of most control systems is the embedded prediction subsystem. This is characterized generically in Figure 19-10. The prediction subsystem takes in a selected control sequence and observable inputs up to the current time T , and produces a prediction of the resulting system response out to some desired $T+\tau$. To accomplish this, the prediction system must contain models that represent all of the complexities required to produce the predicted outcomes *with sufficient accuracy* to support the desired control inputs and desired system outputs. Our focus here is on nonstationary systems that require multi-step prediction.

The Embedded Prediction Component Of A Control System

For nonstationary systems requiring accurate prediction models, one may use discrete event simulation. In this case, the control system produces sets of control sequences to the prediction system and gets back corresponding sets of predicted system responses. The optimal control problem is to come up with the control sequence that meets the *constraints* required of the system while optimizing some prescribed *objective* function.

In the case of interest here, the prediction system requires a simulation, possibly of the type described in the GLOBAL_PLANNER experiment. Many simulations may be required to support optimization of the control sequence to be produced at each real-time clock step when new observation inputs are received. This is an excellent application requiring multiple runs of large-scale simulations within very short time increments.

SUMMARY

As indicated in the above analysis, optimization of a parallel processor software design must be tailored for a particular application. To produce an optimized design, one must account for the many factors described in this book. Although these factors may appear to be considerably different from those typically used for a single processor, they can also simplify single processor architectures while substantially increasing speed.

Parallel processors are used to cut the time to run an application, typically by one or more orders of magnitude. Achieving such speed increases may imply a more significant investment over a single processor design. Once one understands the trade-offs of a particular application, a reasonable estimate of the return on investment can be made, along with a determination of risk. For most applications, whole number multipliers can be achieved with a minimal investment using a 16 or 32 processor PC. Using the CAD system described here, the design and development efforts have proven to be easier than prior efforts using a single processor.

Ensuring fairness of comparisons should not require explicit emphasis. However, it appears that there are many traps one can fall into when making comparisons of parallel processing approaches. For example, if the option exists to buy and use a single processor at a much reduced price, one must compare the fastest parallel processor approach to the fastest single processor for a given application. Similarly, if a Parallel PC can meet the application constraints, huge savings in both time and cost of development and operations may be available. This may make an otherwise unaffordable solution a reality.

CHAPTER 20

NONSTATIONARY APPLICATION ARCHITECTURES

Those who have worked many parallel processor applications, such as those enumerated below, know that each class of applications is very different.

1. Wave Guide simulation
2. Human Body simulation
3. Electro-Magnetic Wave simulation
4. Global HF Power Transmission
5. Global Climate prediction
6. Fluid Flow simulation
7. Biological Particle simulation
8. Chemical - Molecular Structure simulation
9. Scanning, Sorting, and Correlating massive databases
10. Weather Prediction in Mountainous Terrain
11. Power Distribution simulation
12. Global Military Planning simulation

To meet the economic requirements of these applications, one must minimize the cost (number of processors) while meeting run-time performance constraints for a given application.

To solve the technology problem, one must take maximum advantage of the inherent parallelism in each particular application to minimize running time. This requires designing the best space in which to map the inherent parallelism to solve the problem.

In fine-grain problems, e.g., those of molecular structures, and fluid flow, one is typically concerned with the dynamics of particles under the influence of one or more fields, e.g., gravitational, electric, magnetic, pressure and temperature, etc. These systems are typically represented by a large number of cells in a 3D space, such as those used to discretize the system of partial differential equations that represent the smoothed dynamics of the system elements.

Mapping cell blocks in (x, y, z) space into hardware (x', y', z') space is relatively easy, especially when the application space is a rectangular tank. Chapter 19 showed that spaces, e.g., (R, Θ, Φ) , or even sets of different connected spaces are also mapped easily into hardware (x', y', z') space. Wave guides using different complex connected spaces are easily mapped.

To speed up the models of a complex communication space around the globe while maintaining high accuracy, the earth's (LAT, LON, ALT) coordinates are transformed into a large number of sets of (x, y, z) coordinates mapped over the earth's surface to eliminate sines and cosines and thus speed calculations as waves travel through space in straight lines.

Another application that is based on the earth's surface and its surrounding layers is the computation of signal power received on the surface of the earth from an antenna on or near the surface of the earth using transmitters in the HF frequency range. These signals bounce off the ionosphere and can be received far from the transmitter - e.g., on the other side of the earth. These calculations use application spaces similar to those described above. Consequently, they map conveniently into the hardware space.

When approaching fine-grain problems, one is typically concerned with the dynamics of particles under the influence of fields, e.g., gravitational, electro-magnetic, or temperature. Field forces typically depend upon distance from the source of the force, decreasing as $1/R^2$ in most cases. Forces on each particle depend upon those emanating from the other particles. These models are typically described by systems of partial differential equations in three dimensions as well as time, and the state vectors may be large, involving position, velocity, acceleration, etc. When describing particle motion on a computer, one typically uses a discrete time space, ΔT . However, the forces typically move with the speed of light and must be determined at those particles affected - instantaneously - at the end of each ΔT .

But the most difficult parallel processor problem occurs when the connectivity of elements in an application is nonstationary (see Definitions in Appendix A). As described in Chapter 19, there is a connectivity matrix between platforms or cells on each processor that is designed to support the maximum number of platforms or cells of influence around those on a given processor. In very special cases, the connectivity matrix may be nonstationary, making the architecture much more critical. Each of these stationary cases is described below.

Particle Movement

Particles move relative to each other so that at T_1 a particle is influenced by the gravity of one group of particles and at T_N it is influenced by the gravity of another group. The changes generally occur slowly so that loss of particles from the original group and gain of particles to a new group occur - at most - one or a few at each ΔT .

In the case of particle physics, it is the number of particles in each cell within the predefined area of influence around a cell of interest that must be tracked. Since cells do not move with respect to the coordinate system, the connectivity matrix around each cell is fixed, and so is that around a group of cells. It is the changing influence within each cell in the connectivity matrix around a given cell of interest that is changing. In effect, the connectivity matrix around each cell or group of cells simply contains maximum pointer ranges to cells of influence in each direction. Depending on the cell coordinate system, these numbers will likely be the same in each direction for each cell. But unless the coordinate system itself changes, they are stationary.

In the case of particle physics, each particle can be identified by type with specific effects. Thus, organization of the connectivity matrix itself is stationary, and the pointers to the cells containing the changing number of particles are fixed. The most important result is that the connectivity matrix is stationary with respect to processor allocation. As particles move from cell to cell and processor to processor, only the count of particles of a given type in a given cell changes. Their influence always points to the same surrounding cells, and these can all be on the same processor by using copies of that cell information on adjacent processors.

Radio Network Communications On Moving Platforms

Platforms containing radios move relative to each other so that at T1 a platform is connected to one group of elements and at TN it is connected to another group (Note: this is not a cell phone system with connections to base stations that are part of a fixed infrastructure). Changes may occur slowly so that loss of connections from the original group and gain of connections to a new group occur one or a few at each ΔT at most. This depends upon both the speed of the platforms and the surface of the earth (connectivity may change rapidly in mountainous regions).

Moving communication system platforms must be identified and tracked separately. As platforms move, and their connectivity changes, they may be connected to platforms on different parallel processors that are far from their own processor, and no longer connected to those on the same or nearby processors. One must find the best application space architecture to minimize the memory boundary crossing effects caused by this nonstationary (unpredictable) movement.

Of major interest are platforms flying close to the earth's surface, or satellites connected to platforms close to the earth's surface. These may be connected to each other or to stationary (fixed position) platforms on the ground. Or they may be connected to satellites whose paths are fixed and therefore their movement paths are stationary. In this case, platforms whose movement is nonstationary may contain the path databases of those with whom they may be connected and whose movement is stationary with respect to the clock. The calculations of connectivity to these stationary platforms can then be performed directly by the nonstationary platform modules. To pick the best space, application experts must determine the types, quantities, paths and positions of each platform, and how they will be connected.

For example, each type of platform may be equipped with more than one type of radio receiver and transmitter. Each type may have different connectivity depending upon the spectrum, antenna, orientation, signal processing, distance, and environment (terrain, foliage, buildings, noise, etc.). The connectivity of each pair of potential communicators may have to be determined using different models relative to the radio type, and most of these determinations require substantial calculations.

Most of these calculations require knowledge of the surface of the earth and some require properties of the atmosphere and ionosphere surrounding the earth. To anyone having substantial experience in performing these calculations, it is apparent from the types of computation required that the most convenient physical coordinate system is the WGS-84 spheroid representing the earth's surface (the earth is not a sphere). Representation of the environment is important when sufficient accuracy - of Electro-Magnetic wave propagation calculations at RF frequencies of interest - is required. To perform these calculations requires large databases containing detailed data on the earth's environment.

When using this application space to support fast calculations, it becomes necessary to track the positions of the individual platforms with regard to this space. An approach to performing these operations fast on a parallel processor is explained in the next sections.

REPRESENTING COMPLEX RADIO NETWORK COMMUNICATIONS

The most complex communication systems currently in existence are those military radio networks that combine Time Division Multi-Access (TDMA), Frequency Division Multi-Access (FDMA) and Code Division Multi-Access (CDMA) into a single waveform, e.g., the JTIDS / MIDS / LINK16 and EPLRS radio systems. Additional factors contributing to the complexity of these radio networks are enumerated below, refer to Figures 20-1, 2, and 3.

1. All of the network nodes may be moving, with little if any fixed-position infrastructure. Moving platform positions are generally unpredictable relative to the time in a scenario. This is particularly true when decision processes are built into the platforms, so that movement depends upon the changing state of the system, e.g., those based upon C2 inputs, sensor inputs, INTEL inputs and other factors. In other words, the scenario will unfold differently depending upon the relative timing of outcomes. Thus, one must assume that most platform positions are unpredictable, being totally dependent upon the changing scenario as it unfolds. However, satellites follow predictable orbits, so their movement is generally stationary, a property that can be used to simplify the determination of connectivity at the other platforms.
2. Message transmission also depends upon the scenario, so that transmission of messages is also unpredictable, except that messages will be transmitted on the next available time slot allocated for that message. Once the message is put into the queue to be sent, the next time slot for transmission of that message is predictable using TDMA.
3. Each message may use a different FDMA hopping pattern and CDMA coding patterns. These contribute to the determination of probability of reception of the message (PCOM).
4. Each message may use a different power level for the particular message / time slot used for transmission. Interferer/jammer power, temporal and spectral coverage must also be known at the time of reception of the message.
4. The position of each participating platform must be known at the time of transmission and reception, including those of interferers and other noise contributors.
5. The antenna patterns and directions of the participating platforms are important as well as the platform orientations. If an adaptive antenna is available, then the pattern, direction, and orientation of the antenna must be known relative to the platform for the particular time slot used for transmission. This includes patterns as a function of spectral coverage.

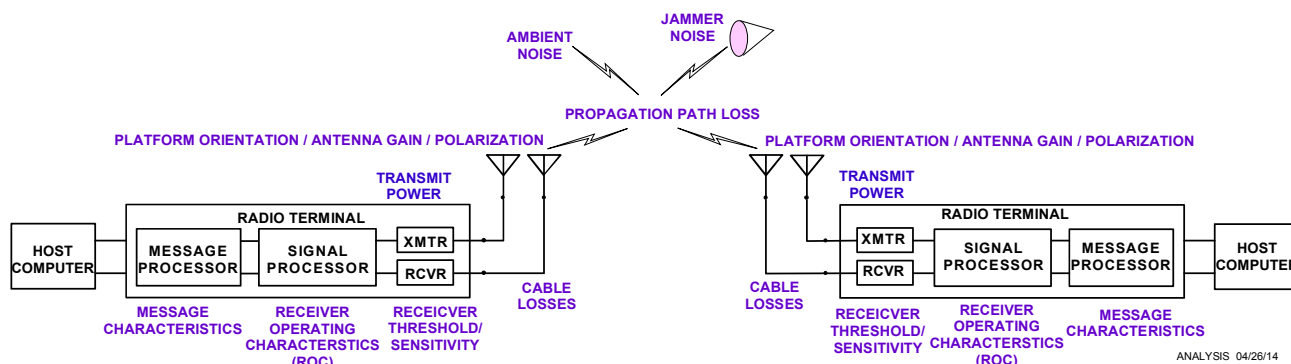


Figure 20-1. Determination of Probability of Communications (PCOM).

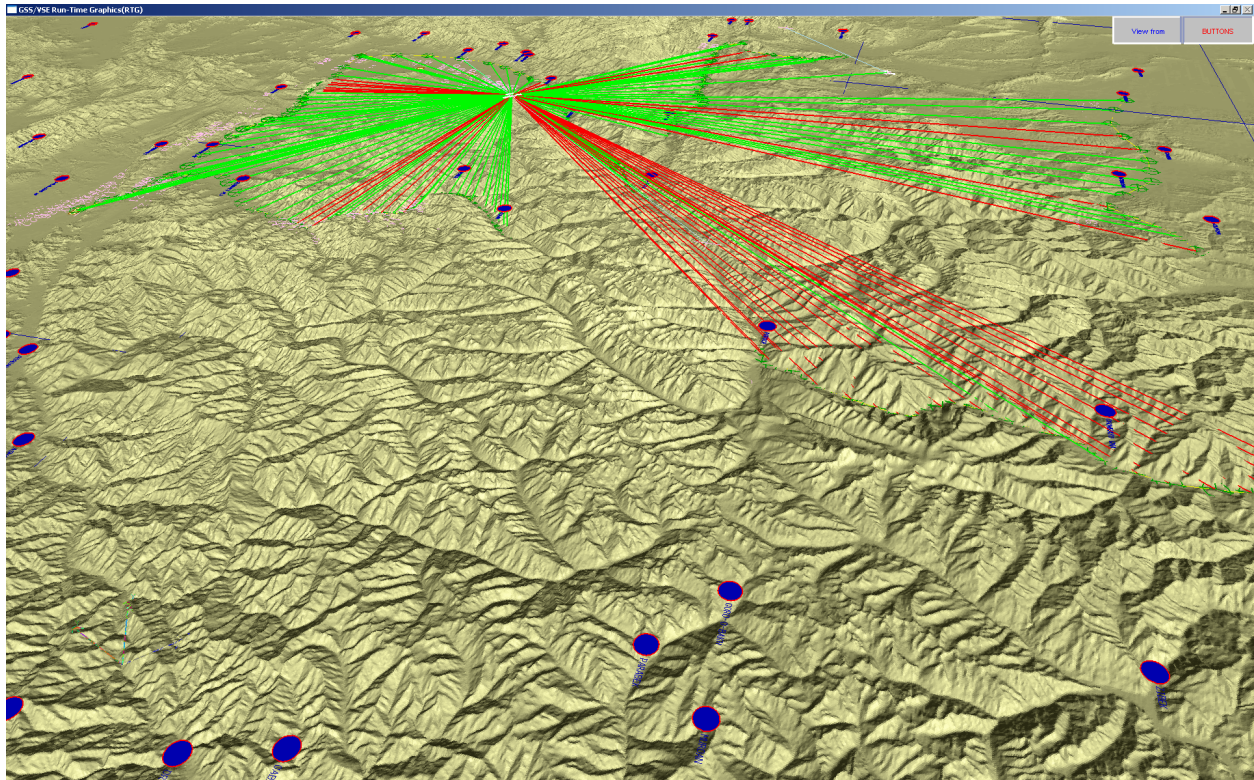


Figure 20-2. Determination of Probability of Communications (PCOM) in mountainous terrain.

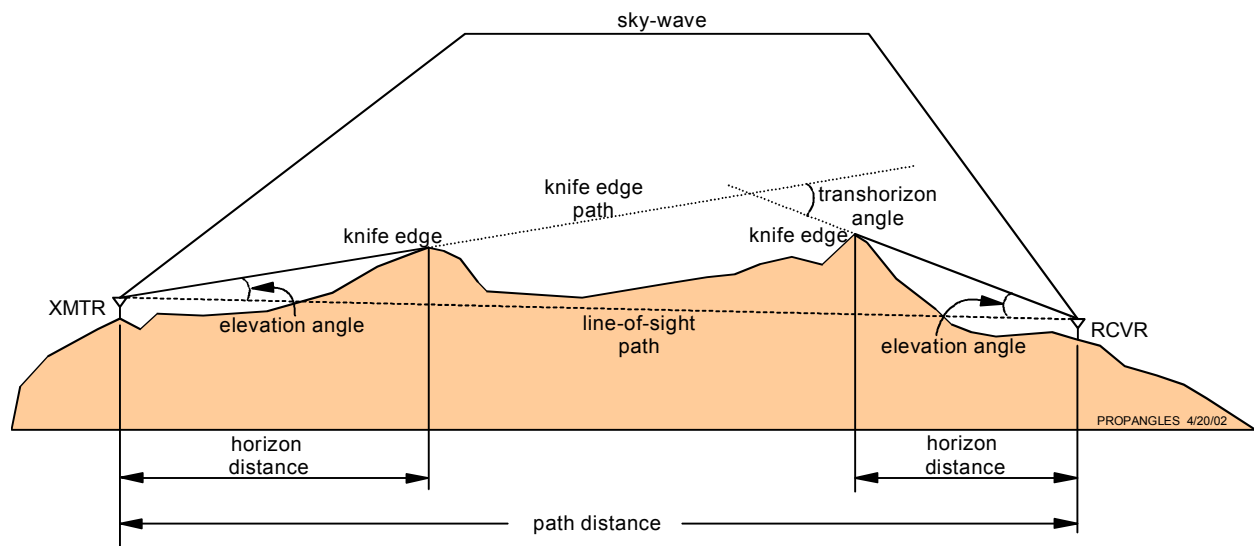


Figure 20-3. Determination of Power at the receiver's antenna.

RADIO NETWORK MODELS FOR PARALLEL PROCESSORS

Given that the space for allocating processors is based upon WGS-84 (LAT, LON, ALT) UTM Zone coordinates transformed into (REL_X, REL_Y, REL_Z) Quad coordinates, the designer is dealing with (REL_X, REL_Y) Quads. Figure 20-4 illustrates a portion of the globe containing on the order of 30 zones. Figure 20-5 illustrates a single REL QUAD corresponding to a northern hemisphere UTM Zone. It is broken into sections to improve algorithm speed.

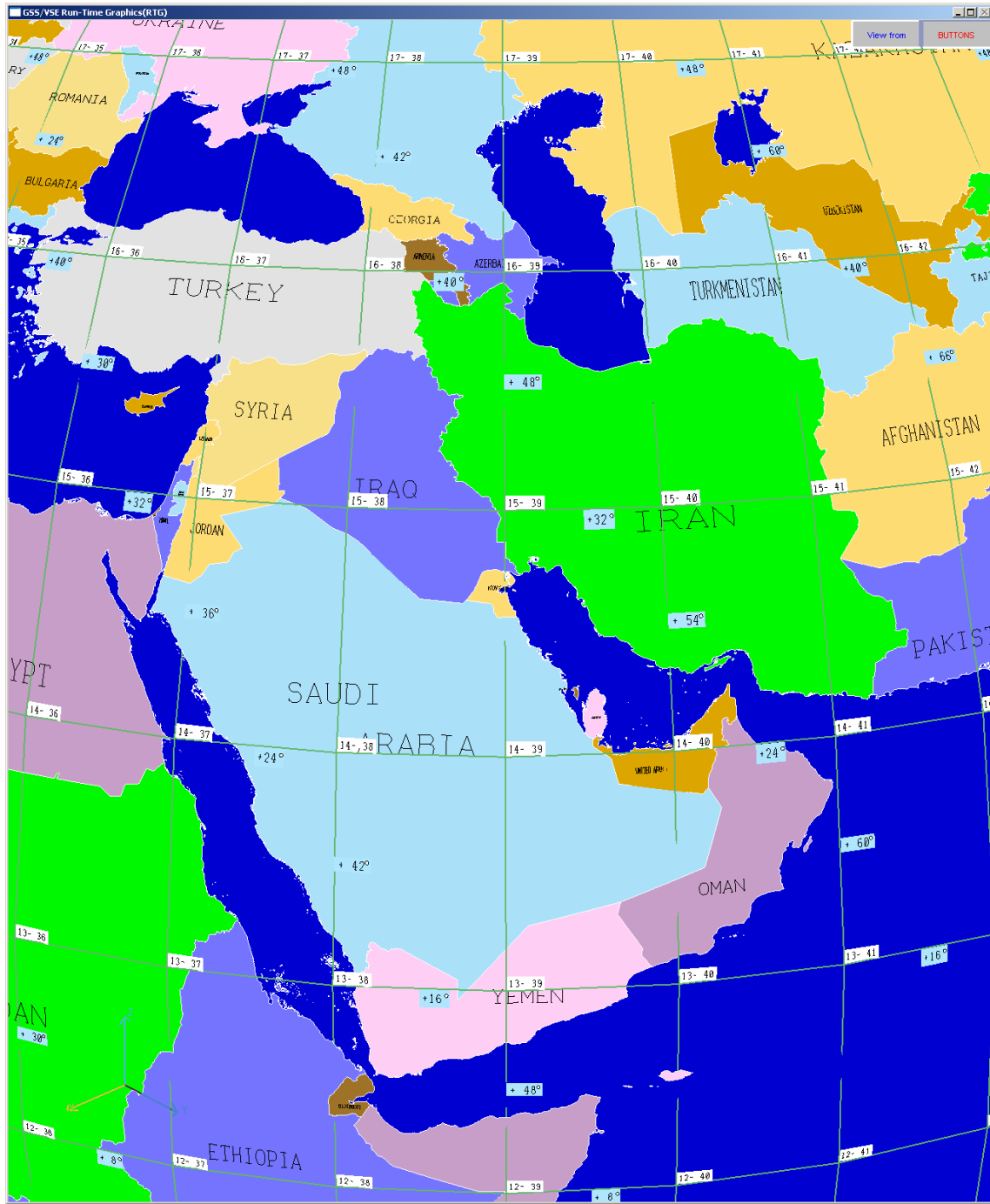


Figure 20-4. UTM Zone boundaries.

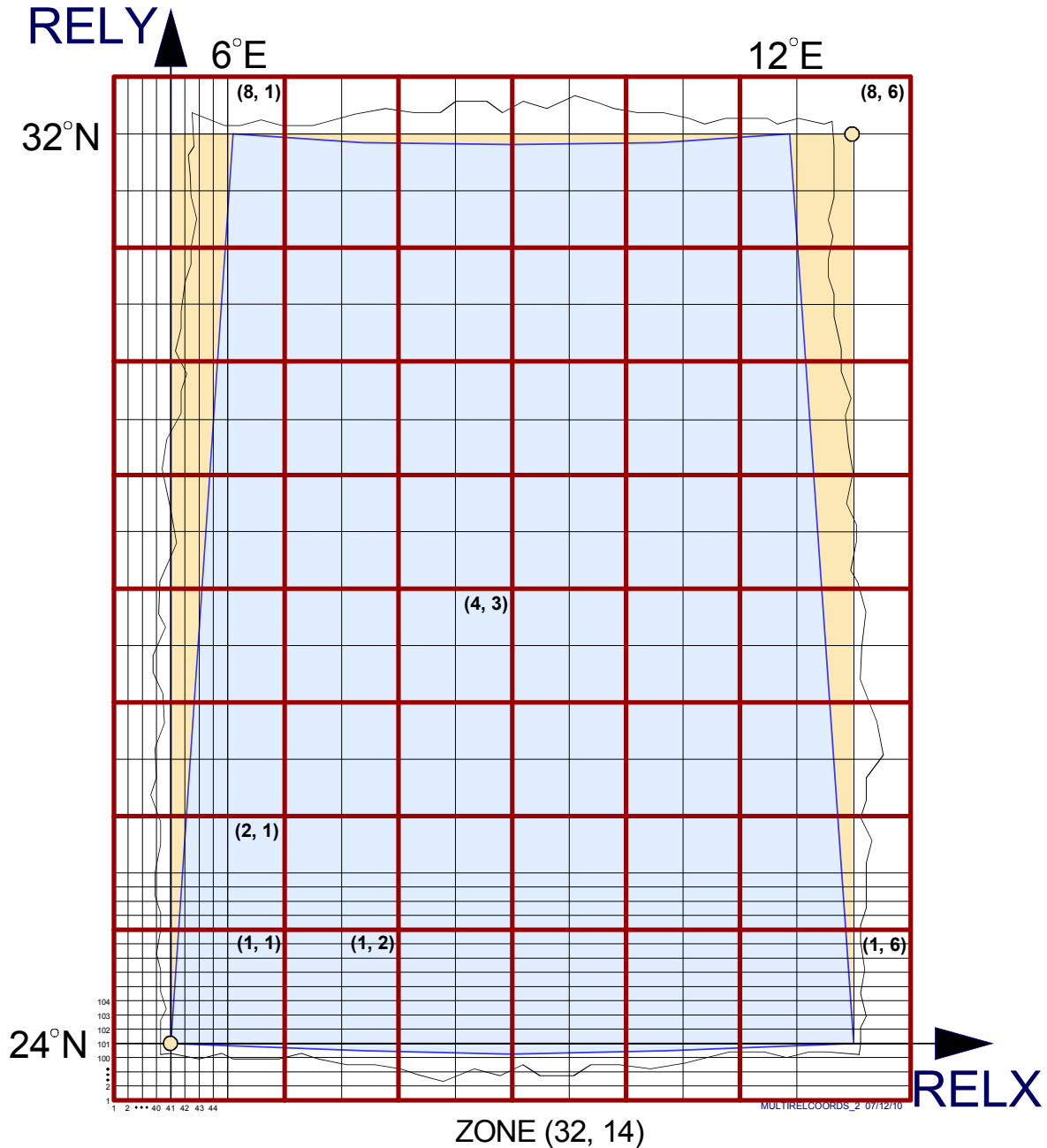


Figure 20-5. Example of a REL QUAD (ZONE) in the Northern Hemisphere.

The above figures illustrate how the globe surface may be decomposed into large standardized mapping areas that overlap. These coordinate systems are currently used to store data describing the earth's surface, e.g., terrain, water, foliage, etc. Figure 20-2 has been produced using extremely fast draw software using terrain data stored in the REL coordinate system.

Hierarchical Platform Databases Mapped Into Hierarchical REL Coordinates

To put the Global Planner simulation onto parallel processors, one can map sufficient numbers of entities to be simulated into each of the designated geographical areas of interest. Each geographical area can be modeled based on the particular scenario sizes in that area. For example, only those Quads containing entities of interest need be modeled. Those Quads of interest that are physically adjacent on the map can be placed similarly on adjacent processors. Quads with fewer entities can be grouped onto a single processor. Space entities, e.g., satellites, may be grouped onto a single processor, or copied onto multiple processors as needed since those models are small. Aircraft generally stay within 20 Km of the earth's surface, so they are related directly to the earth's REL coordinates.

Platform Movement From Quad To Quad

As platforms move within measured time scales (typically not greater than 4 minutes and most often much less) they may cross a Quad boundary. When moving from Quad to Quad, they may be moving from processor to processor. This is easily modeled by transferring copies of their databases to that adjacent processor holding the adjacent Quad.

To accomplish this, the designer must provide enough instances of each platform to cover the maximum that may be active in that processor at any point during the scenario. We note that this is independent of the area of coverage, and only dependent upon the number of platforms that may reside within that area covered by that processor.

Communication Between Platforms In Different Quads

When communications span multiple Quads, one must consider the levels of computation required. If a space platform must communicate with an air or sea platform where terrain is not a factor, then Line-Of-Sight (LOS) must be determined relative to blocking by the earth's surface. LOS determination is done using the earth LOS library utility. This utility will be included in each IND module that needs it. If LOS exists, then messages can be passed between the IND module with the space platform and the air or sea platform.

If the communication is with a platform where terrain may affect the LOS path, then a determination must be made using the positions of both platforms. This is best done by the lowest platform where the data on terrain most likely to interfere is available. The need for additional Quads to be available to that IND module can be determined in advance by the application architect. Other approaches will be much more complex.

Position and other information necessary on the higher platform must be available from that platform. This information is sent to the lower platform to perform the calculation of path loss due to the earth's surface and other environmental factors. Note that path loss can be computed from either end since it is the same in both directions due to the isotropic nature of the media surrounding the earth.

NONSTATIONARY MODEL INTERFACE ARCHITECTURES

To model the dynamic interaction of wireless entities accurately while ensuring that the models are fast computationally, one must analyze the sequence of steps to be taken when critical events occur. This section describes the set of critical events that must be carefully accounted for in a wireless mobile network with all nodes moving. These events must be accounted for in the architecture of the host message processor, transmitter, link, and receiver models in the simulation.

Figure 20-6 provides a map of interactions of transmitters (T1-T12), receivers (R1-R12), and interferers (J1-J4) in different bands (time slot sets). It is used to determine the effects of events of one entity on another as they affect the connectivity of networks sharing the same time slots. Handling these events is critical due to the very dynamic nature of the actual interactions. This is because of the number of potential calculations required as transceivers and interferers turn on and off, change power, and move. Models that affect these actions must be designed to minimize the computational burden of determining the probability that a message has been received correctly.

		Band 1				Band 2				Band 3			
		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Transmitters	Band 1	T1		X	X	X							
		T2	X		X	X					X		
		T3	X	X		X						X	
		T4	X	X	X								X
	Band 2	T5						X	X	X			
		T6					X		X	X			
		T7					X	X		X			
		T8					X	X	X				
	Band 3	T9		X								X	X
		T10			X						X		X
		T11				X					X	X	X
		T12									X	X	X
Interferers	Band 2	J1					X	X	X	X			
		J2					X	X	X	X			
	Band 1	J3	X	X	X	X					X	X	X
		J4	X	X	X	X					X	X	X

GenericArch 9/16/15

Figure 20-6. Effects of RF transmission and jamming on different time slots.

EVENT PROCESSING

Figure 20-7 illustrates multiple transmitters transmitting and multiple receivers listening on the same time slot. The events defined below trigger follow-on events at the link or radio level.

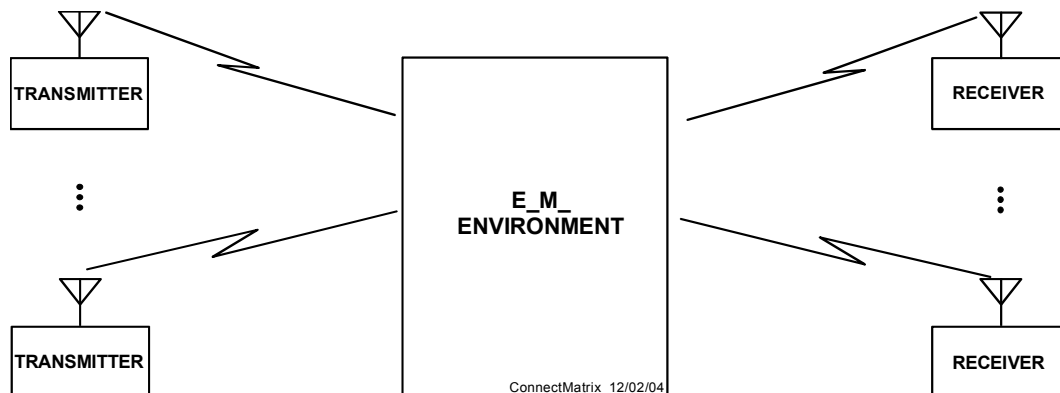


Figure 20-7. Multiple transmitters transmitting and receivers receiving on a given time slot.

Transmitter Turns On, Off, Or Otherwise Changes Power Level

When a transmitter or interferer turns on, off, or changes its power level, the Connectivity matrix (an example column is shown in Figure 20-8) in the E_M_Environment link model must be updated with the information that the transmitter power level has changed, i.e., a new time and a P are entered.

Antenna Moves

When the antenna of an active transceiver moves or changes direction, the link model must be updated with the new position (X,Y, Z) and the new ERP (based upon the position, direction, and orientation of the antenna). The new time must also be put in along with an M (an example column is shown in Figure 20-8).

CONNECTIVITY MATRIX - A SAMPLE COLUMN FOR ONE RECEIVER					
Transmitter Number	Last Update Time/Cause	Last Link Update	Position (X, Y, Z)	Effective Radiated Power (ERP)	Path Loss
1	T1, P	T31	X1, Y1, Z1	dB	dB
2	T2, M	T32	X2, Y2, Z2	dB	dB
3	T3, M	T33	X3, Y3, Z3	dB	dB
4	T4, P	T34	X4, Y4, Z4	dB	dB
5	T5, M	T35	X5, Y5, Z5	dB	dB
6	T6, M	T36	X6, Y6, Z6	dB	dB
...

Figure 20-8. One receiver column in the connectivity matrix.

Transmitter Scheduled To Transmit

When a transmitter is scheduled to transmit on a given time slot, it must enter its transmitter number in the time slot transmission table shown in Figure 20-9 and schedule the receivers that are to receive the message.

TIME SLOT TRANSMISSION TABLE
Number of Transmitters
Trans 1
Trans 2
Trans 3
Trans 4
Trans 5
Trans 6
...

Figure 20-9. Table of transmitters transmitting on a given time slot.

When using parallel processors in a linear homogeneous application, the time steps can be fixed to a maximum time that still ensures sufficient accuracy. In these cases, instead of using the SCHEDULE statement, it is best to use the RELEASE and ACCESS statements.

Receivers Scheduled To Receive

When a receiver is scheduled to receive a message, it must check the time slot transmission table to determine what transmitters it will receive (noise) power from in addition to the one transmitting the desired signal. For each one transmitting, it must then compare the last time its link with that transmitter was updated to the last time that transmitter was updated and determine if and how the link should be updated. If only the power level changed, and not the position of the platform / antenna, received power calculations are not required. If the antenna has only rotated, then only the Effective Radiated Power (ERP) need be recalculated along with the received power. The path loss remains the same

Receivers should only perform those calculations that must be done. If no one has moved, no path loss calculations have to be performed. If only one other transmitter contributing noise has moved, then only that path loss calculation must be done at the time the receiver is to receive.

ADDITIONAL CONSIDERATIONS

There are many other aspects to this application that make it even more difficult. These include multiple types of radio equipment on each platform and how they are connected. They also include human as well as automated decision processes that determine movement and other actions. Finally, end users must run on the order of 100 simulations in short periods of time to perform parameter optimization or parametric analysis to make real-time decisions.

OBSERVATIONS AND SUMMARY

The first page of this chapter lists applications with which the authors are familiar. Of these applications, only the Global Military Planning simulation is known to be nonstationary. Certainly others must exist. What is important is the following:

- It takes substantial application expertise to create the complex application space architectures that are necessary to take advantage of simplified parallel processor architectures. This is true even for the stationary applications listed on the first page.
- VisiSoft provides the ability to map all of these complex application spaces and architectures into simplified parallel processor hardware architectures, especially the nonstationary Global Planner example presented above.
- The running times of applications built using VisiSoft are typically 2 to 4 orders of magnitude faster than those produced using currently popular languages and approaches to building software on parallel processors. These claims have been substantiated by numerous experiments, including many side-by-side comparisons by independent parties on single processors.

Although the authors have worked diligently to present the material provided here in a manner that is relatively easy to digest, this is not a simple technological area. Substantial knowledge is necessary to digest many portions of the material presented. Therefore, readers are encouraged to call VSI (see the inside cover) to talk to the authors to investigate the claims presented, or simply to discuss specific topics or results.

REFERENCES

- [1] Amdahl, G.M., Validity of the single-processor approach to achieving large scale computing capabilities, Proceedings, AFIPS SJCC, Reston, VA, 1967.
- [2] Anselmo, Donald and Henry Ledgard, *Measuring Productivity in The Software Industry*, Communications of the ACM, Vol. 46. No.11, Nov 2003.
- [3] Anselmo, Donald, *Why Software Productivity Has Not Improved*, Software Summit, Washington, D.C., May 2004.
- [4] Anselmo, Donald, *History of the C Programming Language*, Draft, Phoenix, AZ., April 2004.
- [5] ANSI/IEEE Std 1003.1 (ISO/IEC 9945-1) Second edition 1996-07-12, Information technology - Portable Operating System Interface (POSIX®), Part 1: System API (C Language).
- [6] Armour, Phillip G., *Software: Hard Data*, Communications of the ACM, Vol. 49. No.9, Sept. 2006.
- [7] Athans, M. and Falb, P.L., **Optimal Control**, McGraw-Hill, New York, 1966.
- [8] Bailey, D.H., Twelve Ways To Fool The Masses When Giving Performance Results On Parallel Computers, Supercomputing Review, Aug. 1991.
- [9] Baker, F.T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, No 1, 1972.
- [10] Bach, James, "The Immaturity of CMM," American Programmer, Sept., 1994.
- [11] Bach, Maurice J., **Design of the UNIX Operating System**, Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [12] Bell, C.G., "Multis: A New Class of Multiprocessor Computers," Science, Vol. 228, pp 462-467, April 1985.
- [13] Bell, C. G., "Ultracomputers A Teraflop Before Its Time," Communications of the ACM, August 1992, Vol.35, No 8.
- [14] Berry, R., "An Optimal Ordering of Electronic Equations for a Sparse Matrix Solution," IEEE Trans. on Circuit Theory, Vol CT-18, No.1, pp 40-50, January 1971.
- [15] Beyer, Kurt W., *Grace Hopper and the Invention of the Information Age*. Cambridge, MA: The MIT Press, (2009). ISBN 978-0-262-01310-9.
- [16] Booch, Grady, *Software Solutions - Developing the Future*, Communications of the ACM, Vol. 44. No.3, March 2001.

- [17] Bowers, J.C., and Sedore, S.R., *SCEPTRE: A Computer Program for Circuit and Systems Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1971.
- [18] Boehm, Barry, **Software Engineering Economics**, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [19] Brooks, Fred, **The Mythical Man-Month**, Addison-Wesley, Reading, MA, 1975.
- [20] Brooks, Fred, "No Silver Bullets," *IEEE Computer*, April 1987.
- [21] Broy, Manfred, *The "Grand Challenge" in Informatics: Engineering Software-Intensive Systems*, Computer, IEEE Computer Society, 2006.
- [22] *The Unix Timesharing System*, *BSTJ*, Vol. 57 No.6, July- August 1978.
- [23] *The Unix System*, *ATT/BTL Technical Journal*, Vol. 63 No.8, Oct. 1984.
- [24] Business Week, "Is the Computer Business Maturing?", Cover Story, McGraw-Hill, March 6, 1989.
- [25] Business Week, "Can the U.S. Stay Ahead in Software?", Special Software Report, McGraw-Hill, March 11, 1991.
- [26] Business Week, "PROGNOSIS '95", McGraw-Hill, January 9, 1995, pp 72-80.
- [27] Business Week, "Software Made Simple", Cover Story, McGraw-Hill, September 30, 1991, pp 92-100.
- [28] Business Week, Wildstrom, Stephen H., "Price Wars Power Up Quality", *Technology & You*, McGraw-Hill, September 18, 1995, pp26.
- [29] Business Week, *Industry Outlook*, McGraw-Hill, January 12, 2004, pp 92-100.
- [30] Canter, Sheryl, "One-Box Development Systems," *Applications Development*, PC Magazine, July, 1992.
- [31] Cave, W.C. & R.M. Dunn, *Saturation Processing: An Optimized Approach To A Modular Computing System*, ECOM Tech. Rpt 2636, U.S. Army Electronics Command, Ft. Monmouth, NJ, 1965.
- [32] Cave, W., "The Constrained Optimal Design System," *Proceedings IEEE WESCON*, San Francisco, CA, 1971.
- [33] Cave, W., and A. Salisbury, "Controlling the Software Life Cycle - The Project Management Task," *IEEE Transactions on Software Engineering*. Vol SE-4, No 4, July, 1978, pp 326-334.
- [34] Cave, W., and G. Maymon, **Software Life Cycle Management - The Incremental Method**, Macmillan, New York, NY, 1984.
- [35] Cave, W., *Software Survivors*, Software Developer & Publisher, W. Cave, July/Aug1996.

- [36] Cave, W.C., **Simulation of Complex Systems**, Prediction Systems, Inc., Spring Lake, NJ, June 2001.
- [37] Cave, W.C., et.al, The Effects of Parallel Processing Architectures on Discrete Event Simulation, Proceedings: SPIE Defense & Security Symposium, Mar/Apr 2005, Orlando, FL.
- [38] Cave, W. & H. Ledgard, **The Software Survivors**, Visual Software International, Spring Lake, NJ, 2006.
- [39] Cave, W.C., & R.E. Wassmer, Getting Closer to the Machine, Visual Software International Technical Report, Mar 2007, Spring Lake, NJ.
- [40] Cave, W.C., et.al, Estimating Parallel Processing Speed Multipliers, Visual Software International Technical Report, Oct 2007, Spring Lake, NJ.
- [41] Cave, W.C., A Generalized State-Space Framework for Software, Visual Software International Technical Report, Nov 2007, Spring Lake, NJ.
- [42] Cave, W.C., Using Parallel Processors, Visual Software International Technical Report, Feb 2008, Spring Lake, NJ.
- [43] Cave, W.C. & R.E. Wassmer, Understanding Inherent Parallelism, Visual Software International Technical Report, Mar 2008, Spring Lake, NJ.
- [44] Cave, W.C., et.al, **Time is of the Essence: Software Engineering For Parallel Processors**, Visual Software International Technical Report, Oct 2007, Spring Lake, NJ.
- [45] Camford, Richard, "Software Engineering?", IEEE Spectrum, January, 1995, pp 62-65.
- [46] Christensen, Clayton M., **The Innovator's Dilemma**, Harvard Business School Press, Cambridge, MA, 1997.
- [47] Cole, Bernard, EE Times Article, Sep 13, 2007, www.eetimes.com/showArticle .
- [48] Constantine, Larry, "Back to the Future," Communications of the ACM, Vol 44, No. 3, March, 2001.
- [49] Culler, D.E. & Singh, J., **Parallel Computer Architecture: A Hardware/Software Approach**, Lib. Cong. #QA76.58.C85, *Morgan-Kaufmann*, 1999.
- [50] Cusumano, Michael A., "What Road Ahead for Microsoft and Windows?," Communications of the ACM, July 2006, pg 21-23.
- [51] Daly, W., "The J-Machine: A Fine-Grain Concurrent Computer; MIT VLSI Memo 89-532, May, 1989.
- [52] Daconta, Michael C., *C⁺⁺ Pointers and Dynamic Memory Management*, John Wiley & Sons, NY, 1995.

- [53] Deming, W. Edwards, *Out of the Crisis*, MIT CASE, Cambridge, MA, 1992.
- [54] DeMarco, Tom, **Controlling Software Projects**, Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1982.
- [55] DeMarco, T and T. Lister, **Peopleware**, Dorset House, New York, NY, 1987.
- [56] Ferguson, Steve, *The History of Computer Programming Languages*,
http://www.princeton.edu/~ferguson/adw/programming_languages.shtml
- [57] Fitzsimmons, A., and T. Love, "A Review and Evaluation of Software Science," ACM Computing Surveys 10, No. 1, March 1978, pp 3-18.
- [58] Frank, S., "Tightly Coupled Multiprocessor System Speeds Memory Access Times," Electronics, pp 164-169, 1984.
- [59] Gauthier, Richard, and Pont, Stephen. **Designing Systems Programs**, Prentice-Hall, Englewood Cliffs, N.J., 1970.
- [60] Gelb, A., Editor, **Applied Optimal Estimation**, MIT Press, Cambridge, MA, 1974.
- [61] Gilder, George, **Microcosm**, Simon and Shuster, New York, NY, 1989
- [62] Glass, R.L., *Looking into the Challenges of Complex IT Projects*, Communications of the ACM, Nov 2006.
- [63] Gordon, G., *A General Purpose Systems Simulation Program*, Proc. EJCC, Washington, D.C., pp 87-104., MacMillan Publishing Co., New York, 1961.
- [64] Gordon, G., **The Application of GPSS V to Discrete System Simulation**, Prentice Hall, Englewood Cliffs, NJ, 1961.
- [65] Gordon, J., **System Simulation**, Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [66] Groth, R., *Is the Software Industry's Productivity Declining?*, IEEE Software, Nov/Dec 2004.
- [67] *GSS User's Reference Manual*, Version 10.6, Visual Software International, Spring Lake, NJ, 1984-2013.
- [68] Gustafson, J.L., Reevaluating Amdahl's Law, CAACM, 31(5), May 1988.
- [69] Guth, Robert A., "Battling Google, Microsoft Changes How It Builds Software," The Wall Street Journal, Sept 23, 2005, page 1, column 5.
- [70] Hactel, G.D. and Roher, R.A., *Techniques For The Optimal Design And Synthesis Of Switching Circuits*, Proceedings of the IEEE Special Issue on CAD, Nov 1967, pp 1864.
- [71] Hactel, G.D. et al, *The Sparse Tableau Approach To Network Analysis And Design*, IEEE Transactions on Circuit Theory, Jan 1971, pp 101.

- [72] Hafner, Eric, *Theory and Design of Oscillators*, Proceedings of the IEEE, New York, NY, in two issues, 1977
- [73] Hendrickx, David S., Simulation Systems: A New Role in Executive Decision Making, *Signal*, Vol 42, No. 1, July 1988.
- [74] Holland, J.H., "A Universal Computer Capable Of Executing An Arbitrary Number Of Sub-Programs Simultaneously," *Proc EJCC*, pps 108-113, 1959.
- [75] INFOWORLD, "DILBERT WOULD RELATE," *Managers Bulletin Board*, February 6, 1995, pp 62.
- [76] INFORM, "Calling all COBOL users," *Digital Equipment Corp.*, Sept./Oct. 1995, pp 1.
- [77] Jones, Capers, **Applied Software Measurement: Assuring Productivity and Quality**, McGraw Hill, New York, NY, 1991.
- [78] Jones, Capers, "Evaluating International Software Productivity Levels," Version 3.0, Software Productivity Research, Inc., Burlington, MA, July, 1991.
- [79] Juran, Joseph M., **Juran's Quality Handbook**, Fifth Ed., McGraw-Hill, 1999.
- [80] Yasushi Kambayashi and Henry F. Ledgard, "The Separation Principle - A Programming Paradigm" *IEEE Software*, March/April 2004
- [81] Kernighan, B.W., and D.M. Ritchie, **The C PROGRAMMING LANGUAGE**, Prentice Hall, Englewood Cliffs, NJ, 1973.
- [82] Kleiman, Steve, DeVang Shah, Bart Smaalders. *Programming with Threads*, Sunsoft Press, Sun Microsystems, Mountain View, CA 1996.
- [83] Koniges, Alice E., (Editor), **Industrial Strength Parallel Computing**, Lib. Cong. #QA76.58.1483, *Morgan-Kaufmann*, 2000.
- [84] Krishnadas, L.C., EE Times Article, Jan 17, 2008, http://www.eetimes.com/document.asp?doc_id=1167793 .
- [85] Kuhn, Thomas, **The Structure of Scientific Revolutions**, The University of Chicago Press, Chicago, IL, 1970.
- [86] Kumar, M., Measuring Parallelism in Computation-Intensive Scientific.Engineering Applications, *IEEE Transactions on Computers*, Vol 37, Issue 9, Sep 1988.
- [87] Dr. Anita J. La Salle, *Software Industry and Economic Security*, Software Industry Workshop, April 27, 2000.
- [88] Ledgard, H., et al, "The Natural Language of Interactive Systems," *CACM* No. 10, October 1980, pp 556-563.
- [89] Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Oct 2000.

- [90] Ledgard, H.F. et.al, Language Properties To Improve Productivity And Speed, University of Toledo, Aug 2007.
- [91] Ledgard, H.F., The State of the Software Industry, University of Toledo, Aug 2007.
- [92] Levy, Leon, **Taming the Tiger: Software Engineering and Software Economics**, Springer-Verlag, New York, NY, 1987.
- [93] Mahoney, Michael S. *The Unix Oral History Project*,
<http://www.princeton.edu/~mike/expotape.htm>
- [94] Maslo, R. and W.C. Cave, A New Approach to Development and Support of Real-Time Control Systems, Proceedings, 7th Annual GSS User's Conference, Prediction Systems, Inc., Spring Lake, NJ, June 1994.
- [95] Marcotty, Michael, **Software Implementation**, Prentice Hall, New York, NY, 1991.
- [96] McClure, R.M., *TMG - A Syntax Directed Compiler*, Proc. 20th ACM NationalConference, 1965.
- [97] Meindl, James D., Keynote Address, 1997 Hot Chips IX Symposium, Stanford Un., Palo Alto, CA.
- [98] Merritt, R., Wintel will fund parallel software lab at Berkeley, EE Times On Line, Feb 13, 2008, www.eetimes.com/showArticle.jhtml?articleID=206503988 .
- [99] Merritt, R., Multicore puts screws to parallel programming modules, EE Times On Line, Feb 15, 2008, <http://www.embedded.com/design/mcus-processors-and-socs/4023225/Multicore-puts-screws-to-parallel-programming-models> .
- [100] Merritt, R., Berkeley researcher describes parallel path, EE Times On Line, Feb 21, 2008, www.eetimes.com/showArticle.jhtml?articleID=206801376 .
- [101] Merritt, R., Opinion: Time to plow multiple paths to parallel computing, EE Times On Line, Feb 22, 2008, www.eetimes.com/showArticle.jhtml?articleID=206801229 .
- [102] Mills, H.D., Mathematical Foundations of Structured Programming, Technical Report FSC 72-6012, IBM Federal Systems Division, 1972.
- [103] Mitchell, R., "In Supercomputing, Superconfusion," Business Week, pps 89-90, March, 1993.
- [104] Netizens: An Anthology, Chap 9, *On the Early History and Impact of Unix - Tools to Build the Tools for a New Millennium*, <http://www.columbia.edu/~rh120/ch106.x09>
- [105] Pargus, R. and P. Kambekar, "Guidelines for Dynamic Load Balancing in Conservative Distributed Simulations," Proceedings of the 1990 Winter Simulation Conference, New Orleans, LS, pp 47-52.
- [106] Parnas, D., "Education for Computer Professionals," IEEE Computer, January 1990, pp 17-22.

- [107] Parnas, D. L. On the Criteria To Be Used in Decomposing Systems into Modules. Comm. ACM 15, 12 (December, 1972), 1053-1058.
- [108] Patterson, D., Bell, G., et al, "Massively Parallel Uproar," Upside, pps 88-97, March, 1992.
- [109] Perkins, T., and R. Kalgaard, "Inside Upside," Upside Magazine, September 1991.
- [110] Pfleeger, S. L., "Viewpoint: Software Engineering Needs to Mature", IEEE Spectrum, January, 1995, pp 64.
- [111] Poore, Jesse H., *A Tale of Three Disciplines and a Revolution*, IEEE Computer Society, Jan 2004.
- [112] *Visual Software Development For Parallel Machines*, Final Report, US Army CECOM Contract DAAB07-97-C-H501, Prediction Systems, Inc., Spring Lake, March, 1997.
- [113] *Multi-Computer Version of GSS*, Final Report, DARPA MHPCC BAA Consortium, Prediction Systems, Inc., Spring Lake, NJ, Sept. 1998.
- [114] *High Efficiency, Scalable, Parallel Processing*, DARPA Contract SF022-035 Final Report, Prediction Systems, Inc., Spring Lake, NJ, June 2003.
- [115] Ramadge, P.J. and W.M. Wonham, "Supervisory Control of a class of discrete-event processes," SIAM J. Control Optimization, vol 25, no.1, pp 206-230, Jan. 1987.
- [116] Ramadge, P.J. and W.M. Wonham, "The Control Of Discrete-Event Systems," Proc. IEEE, vol 77, no.1, pp 206-230, Jan. 1989.
- [117] Ranum, Marcus J., *SECURITY - The root of the problem*, ACM QUEUE, June 2004.
- [118] Richards, M., *BCPL: A tool for Compiler Writing and Systems Programming*, Proc. AFIPS SJCC 34, 1969.
- [119] Ritchie, Dennis, *Evolution of the Unix Time-sharing System*, ATT/BLTJ, Vol. 63 No.8, Oct. 1984.
- [120] Richie, Dennis M., *Development of the C Language*, Second History of Programming Conf., 1993, <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>
- [121] Rieher, P., "Parallel Simulation Using the Time Warp Operating System," Proceedings of the 1990 Winter Simulation Conference, New Orleans, LS, pp 38-45.
- [122] Roosta, Seyed H., *Parallel Processing & Parallel Algorithms: Theory & Computation*, Lib. Cong. #QA76.58.R66, Springer, 1999.
- [123] Rose, F., and R. Turner, "A Jungle Out There", Front Page Article, The Wall Street Journal, January 23, 1995.
- [124] Rosen, J.P., "What Orientation Should Ada Objects Take?" CACM, Vol 35, No 11, November 1992, pp 71-76.

- [125] *RTG User's Reference Manual*, Version 4.2, Prediction Systems, Inc., Spring Lake, NJ, 1994.
- [126] Rubin, K.S., "Reuse in software Engineering: An Object Oriented Perspective," Proceedings of IEEE COMPCON, Spring, 1990.
- [127] Santos, Robert, "Simulation to Support Real-Time Control of Communications Networks", Proceedings of the Society for Computer Simulation Western Multiconference, Anaheim, CA, January 1991.
- [128] Schweppe, F., **Uncertain Dynamic Systems**, Prentice Hall, Englewood Cliffs, NJ, 1978.
- [129] Carnegie Mellon University - Software Engineering Institute (SEI) - Capability Maturity Module (CMM) , www.sei.cmu.edu/cmm .
- [130] Shannon, C.E., a mathematical theory of communication, BSTJ, Vol.27, pp 379 & 623, Jul & Oct 1948.
- [131] Sherr, A.L., "Developing and Testing a Large Programming System," In *Program Test Methods*, W.C. Hetzel, Editor, Prentice-Hall, Englewood Cliffs, NJ, 1972.
- [132] Shi, Y., Reevaluating Amdahl's Law and Gustafson's Law, Temple Un., Oct 1996.
- [133] Shore, David, "Computer Technology Stands Poised for Substantial Leap," Signal, February 1993, pp 35-37.
- [134] Sietz, C., "The Cosmic Cube," Communications of the ACM, 28-1, pps 22-23, January 1985.
- [135] Singleton, R.C., "An Efficient Algorithm for Sorting with Minimum Storage," CACM, Vol 12, No 3, March 1969, pp 185-187.
- [136] Sitner, Jerry, "Viewpoint - How Much Longer," Mainframe Journal, July 1990, pp 120.
- [137] Standish Group International, "Chaos, Charting the Seas of Information Technology," The Standish Group International Inc., Report, Dennis, MA, 1995.
- [138] Standish Group International, "Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%," Press Release, The Standish Group International Inc., West Yarmouth, MA, 2003. <http://www.standishgroup.com/press/article.php?id=2> .
- [139] Standish Group International, "Standish: Project Success Rates Improved Over 10 Years" Software Magazine, January 2004. <http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish>
- [140] SFI Files, *A Description of the Standard File Interface*, Version 2, Prediction Systems, Inc., Spring Lake, NJ, 1995.
- [141] Strassmann, Paul, "From a Craft to an Industry," Ada Symposium, George Mason University, 1992.

- [142] Stroustrup, B., "What is Object-Oriented Programming?" IEEE Software, May 1988, pp 10-20
- [143] Stroustrup, B., "What is Object-Oriented Programming? (1991 revised version). Proc. 1st European Software Festival. February, 1991 - <http://www.public.research.att.com/~bs/whatis.pdf> .
- [144] Sutter, Herb, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency In Software, Dr. Dobbs's Journal, 30(3), March 2005.
- [145] Sutter, Herb, and J. Larus, Software and the Concurrency Revolution, ACM Queue, vol. 3, no.7, September 2005
- [146] Upside Magazine, "Musings on the Millenium," Feature Editorial, October 1994.
- [147] van der Linden, P, **Expert C Programming - Deep C Secrets**, SunSoft Press - Prentice Hall, Englewood Cliffs, NJ, 1994.
- [148] Von Neumann, J, "First Draft of a Report on EDVAC", Moore's School of Electrical Engineering, University of Pennsylvania, June 30, 1945.
- [149] *VisiSoft GENERAL Library & RTG_DRAW Library*, Software Description Document, Visual Software International, Spring Lake, NJ, 2006.
- [150] *VSE User's Reference Manual*, Version 10.4, Visual Software International, Spring Lake, NJ, 2005.
- [151] VSI Engineering Technical Report: *The Parallel PC*, Visual Software International, Spring Lake, NJ, 2013.
- [152] Wakabayashi, Daisuke, "Microsoft's Top Visionary Sees a Paarallel World", Reuters Article, Seattle, 03/13/08, <http://in.reuters.com/article/2008/03/13/microsoft-mundie-idINN1222563020080313> .
- [153] Weinberg, G., **An Introduction to General Systems Thinking**, John Wiley & Sons, NY, 1975.
- [154] Wilbur, M., **Managing Software Reliability: The Paradigmatic Approach**, North Holland, New York, NY, 1981.
- [155] Yourdon, E, **Decline & Fall of the American Programmer**, Yourdon Press - Prentice Hall, Englewood Cliffs, NJ 1993.
- [156] Zadeh, L.A. and Desoer, C.A., **Linear System Theory: The State Space Approach**, McGraw-Hill, NY 1963.
- [157] Zuniga, Gilberto, Concepts for the Implementation of an Air Defense Model Base, Proceedings of the 55th Military Operations Research Symposium, May 1987
- [158] NSF Program Solicitation NSF 14-516, Exploiting Parallelism and Scalability (XPS), Feb 2014 <http://www.nsf.gov/pubs/2014/nsf14516/nsf14516.pdf> .

APPENDICES

APPENDIX A - Definitions

Ever since Moore's curve leveled off, applications bound by significant computer speed increases are forced to rely on parallel processing. This supposedly difficult task becomes simple when using VisiSoft application spaces to transform software architectures into hardware architectures. Just by separation of applications into the two hardware environments defined below, the hardware architectures for each are simplified. Specifically, parallel processor applications are much different from those of a server. Those applications requiring both capabilities are easily split into separate software architectures. Mapping a speed-bound application space architecture into a good parallel processor architecture is simple using VisiSoft.

Server Systems

These typically support large *I/O Bound* applications requiring communication networks for transaction processing, and database access and management. A large server system must interface with many I/O facilities, including networks of workstations and big disk management. Servers are composed of large numbers of processors, where each processor typically runs multiple tasks with fat communication channels to fast I/O, including teleprocessing channels and big disk facilities. Applications include large commercial data processing, huge database management including query, and remote teleprocessing for cloud type applications. Given enough processors, it can also support embarrassingly parallel applications defined below.

Parallel Processors

These are required to support true *Speed Bound* applications that are not embarrassingly parallel (definition below). Parallel processor applications have substantial inherent parallelism. These parallel elements can be put into independent modules that influence each other but can run in parallel on a large number of processors to meet the time constraints for a single task. They require limited one-way I/O (typically initialization prior to running, and output during and after running). They typically require intensive internal processing of large mathematical systems or decision processes that are processed in parallel. Examples of parallel processor applications are EM wave simulation, meteorological simulation, and fluid dynamic simulations (e.g., fluid flow through multiple container surfaces; moving particle physics; and dynamic biological, and chemical particle interactions, etc.).

Embarrassingly Parallel Applications

These may be broken into multiple separate tasks running on separate processors. Once they start to run, they need little if any communication between processors. They may run sufficiently fast on a server with many processors or on a cluster of PCs. Scientific applications, e.g., Monte Carlo simulation and fast approaches to large scale Linear Programming (LP) are embarrassingly parallel. These applications are poor examples of the requirements for real parallel processors applications.

Linear Versus Nonlinear Applications

When using parallel processors in a linear application, the application time steps (may be different from the VPOS ΔT) can be set to a maximum time that still ensures sufficient accuracy for a particular module. In nonlinear applications, a subset of modules or instances typically use smaller time steps to achieve sufficient accuracy when operating in nonlinear regions. Other modules will be waiting while those in the nonlinear region converge and complete. This conserves time when modules are on the same processor. These time differences can be achieved using SCHEDULE statements.

Homogeneous Versus Nonhomogeneous Applications

Homogeneous implies that all IND modules perform functions with each ΔT . In nonhomogeneous applications, a subset of the IND modules does not perform any functions within a potential string of ΔT s. In homogeneous applications, Cross-SCHEDULE statements are generally unnecessary and it is best to use the SET EVENT, WAIT UNTIL, RELEASE, and ACCESS statements. Nonhomogeneous applications are typically influenced by outside factors or events that are best invoked using the SCHEDULE statement.

Stationary Versus Nonstationary Applications

In stationary applications the connectivity matrix of application modules or module instances remains constant. In nonstationary applications the connectivity matrix of modules or module instances changes. This causes access data transfer delays to increase unless the module databases are moved to different processors to maintain an optimal mapping of the diagonalized application connectivity matrix with respect to the processor connectivity matrix. In these cases, copies or instances of the module can remain stationary on those processors that use them. Only the databases need be moved.

APPENDIX B - Process & Resource Size Statistics

To ensure best use of memory space for speed, it is desirable to break large resources into pre-specified segment sizes, and to bring them in as they are used. To make decisions on segment sizes, one must analyze the statistics of cutoff sizes when loading resources into memory. Figure B-1 shows a statistical distribution of process sizes from a sample of models. All processes fit within a single page. The largest takes slightly more than a half page. Figure B 2 shows a statistical distribution of resource sizes from a sample of models. It is a bi-modal distribution with most resources being far less than a page and one under two pages. The other mode has larger resources that are spread from 100 Kb to well over 100 Kb.

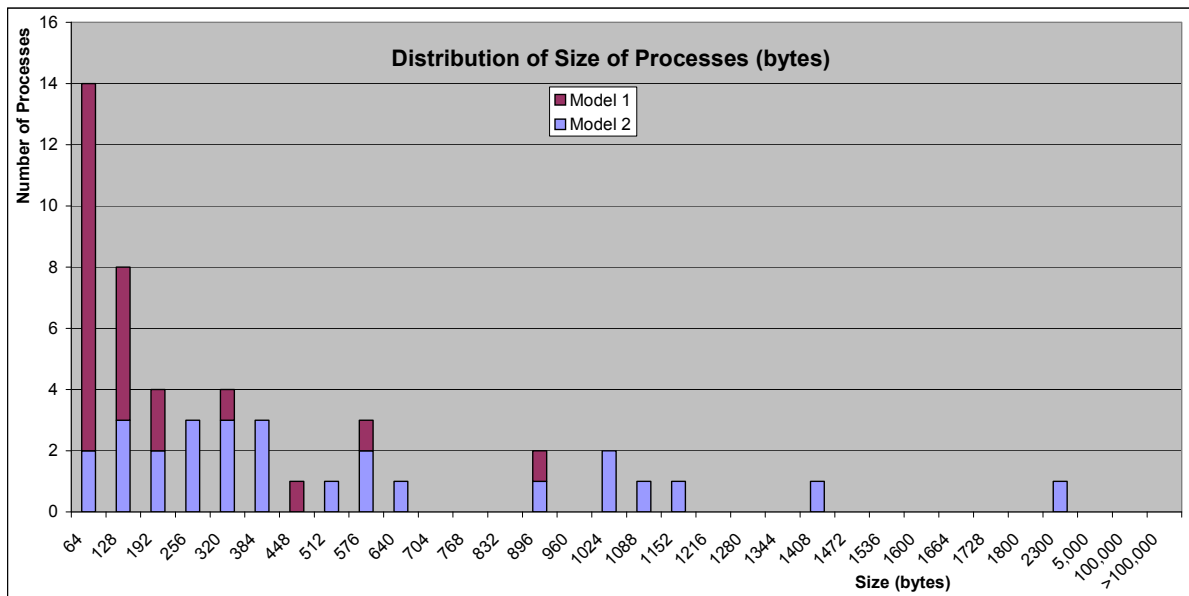


Figure B-1. Statistical distribution of process sizes.

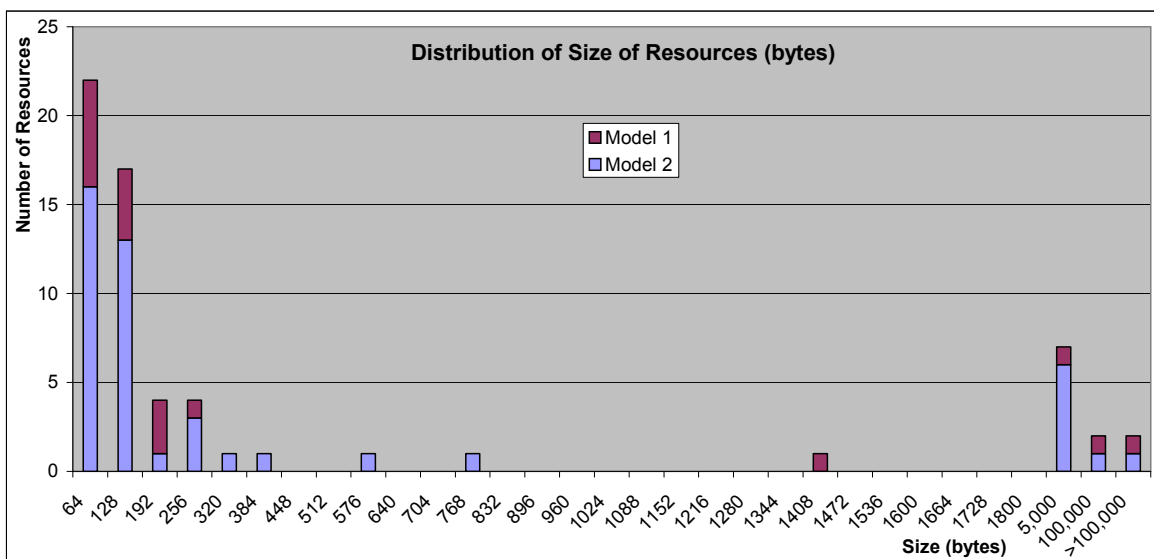


Figure B-2. Statistical distribution of resource sizes.

Additional Process Size Statistics

The following is an analysis of instruction memory utilization for the Joint Airborne Network Control (JANC) simulation. Figure B-3 represents the main JANC modules. It must be qualified with the following points.

- It contains all of the library modules and utility modules.
- It represents the entire JANC set of modules - which includes the initialization facilities and interactive graphical facilities - that are not part of the instanced IND Modules. It contains all of the instanced models (as opposed to a single instance).
- It is a static representation of memory by process and does not represent the amount of time spent running that process.

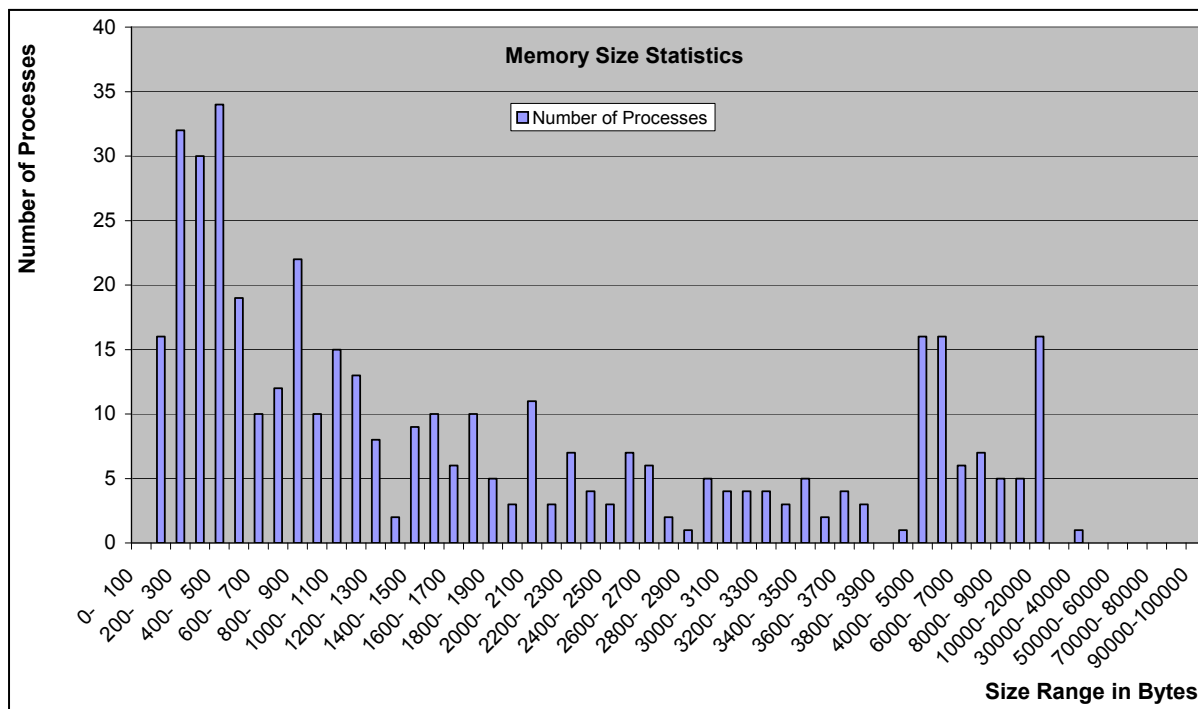


Figure B-3. Distribution of instruction memory utilization for JANC.

Summary Statistics:

TOTAL USER PROCESSES:	417
TOTAL PROCESS SIZE (Bytes):	1,020,656 - 256 pages
AVERAGE PROCESS SIZE (Bytes):	2,447 - 2+ pages
MEDIAN PROCESS SIZE (Bytes):	1,200 - 1+ page

APPENDIX C - Expanded Memory Maps For Better Use Of Chip Space

The following tables indicate how cache memory could be expanded with the removal of hardware memory management techniques which are unnecessary when using VisiSoft. These breakpoints are best determined using simulations of various applications. This can be done by taking in the source code for a given application and creating a pseudo-assembly code that is used by parameterized models of the internal hardware processor and resulting speeds.

N	2**N BYTES			
0	1			
1	2			
2	4			
3	8			
4	16			
5	32			
6	64			
7	128			
8	256			
9	512			
10	1,024	KILO		
11	2,048			
12	4,096			
13	8,192			
14	16,384			
15	32,768			
16	65,536			
17	131,072			
18	262,144			
19	524,288			
20	1,048,576	MEGA		
21	2,097,152			
22	4,194,304			
23	8,388,608			
24	16,777,216			
25	33,554,432			
26	67,108,864			
27	134,217,728			
28	268,435,456			
29	536,870,912			
30	1,073,741,824	GIGA		
31	2,147,483,648			
32	4,294,967,296			
33	8,589,934,592			
34	17,179,869,184			
35	34,359,738,368			
36	68,719,476,736			
37	137,438,953,472			
38	274,877,906,944			
39	549,755,813,888			
40	1,099,511,627,776	TERA		
41	2,199,023,255,552			
42	4,398,046,511,104			
43	8,796,093,022,208			
44	17,592,186,044,416			

1 PAGE			
16 PAGES	L1 Instruction CACHE	1 Processor	
64 PAGES	L1 Data CACHE	1 Processor	
1024 PAGES			
4096 PAGES	L2 Chip CACHE	18 Processors / Chip	
L3 CACHE	256 X L2 CACHE	72 Processors / Box	
BOX RAM	128 X L3 CACHE		
Platform RAM	32 X BOX RAM	32 Boxes - 2304 Processors	

Figure C-1. VisiSoft Memory Mapping Table.



Software Theory for Parallel Processors

The technology described in this book is a revelation in software, a field that lost its scientific direction over three decades ago. Too many engineering people have dropped out of the field because of the lack of experimentation and measurement needed to improve quality, productivity, and run-time speed. To the best of my knowledge, this is the first publication that provides a sound scientific basis for improving these measures and heading the software field in the right direction.

Bob Santos, BSEE/MIT, formerly Sr. Vice Pres AT&T, Basking Ridge, NJ.

This book explains the theory and application of a totally new approach to building software. Without this approach, the software field is clearly stuck in a rut compared to hardware approaches being developed by engineers to build computers today, particularly those with multiple processors. This will become apparent as more processors are put into computers to speed up run times.

Ron Maslo, Ph.D., P.E., formerly with AT&T Bell Laboratories, Holmdel, NJ.

I have tracked the technology described here since its early beginnings and am familiar with many of the huge simulations built for the military using it. I am not aware of a technology that comes close to this for productivity, as well as maintaining high quality and tight control over extremely large complex software systems.

Alan Salisbury, Maj Gen U.S. Army (ret.), Ph.D. EE/Stanford

If you want to know where the software field is headed in the next three decades, read this book. It is the most significant innovation in software since the compiler. The fact that it takes an engineering background to understand the hard science underlying the concepts should no longer be of concern. In fact, use of the CAD system it describes can be learned at the high school level as well as by subject area experts who want to build their own software.

Jerry Tuttle, VADM US Navy (ret.)

IETC Publications, Spring Lake, NJ 07762