Application Space Architecture (ASA) - The ISA For Parallel Processors

W.C. Cave & R.E. Wassmer† - November 19, 2015

"It is easier to write a new code than understand an old one."
John von Neumann

Background

The first *stored-program* computer, the MANIAC, was developed at the Institute for Advanced Studies at Princeton University in the early 1950s. Its design was determined by John von Neumann's Instruction Set Architecture (ISA), to implement a new concept described by Eckert and Mauchly following their previous design of the first all-electronic tabulator, the ENIAC, at the University of Pennsylvania. Von Neumann's ISA defined a set of binary instructions needed to write computer programs to implement a wide range of applications, from weather prediction to solving ballistic equations, and the compression equations required to produce enough energy to set off a nuclear reaction. He did not design the MANIAC. He provided the stored-program requirements (the ISA) to those who did the logical design to implement his binary instruction set. Clock rates were determined by vacuum tube switching speeds at the time, and memory size quickly became the major factor for improving speed.

Starting with the Holland machine in 1960, many organizations claimed to have a "breakthrough" in digital computer design to replace the von Neumann architecture. Upon reading the history, one finds that they all failed to succeed when faced with a variety of applications. What they did not understand is that von Neumann's main contribution was the ISA required to meet a broad range of application requirements. These other designs did not follow from a well-defined ISA.

Single processor ISAs have evolved over the years, but these designs only process instructions sequentially, effectively following Alan Turing's original tape processor design, [8]. This sequential processing scheme can be mapped into a one dimensional spatial architecture, with binary instructions stored in memory locations numbered from 1 to N, where N has become a very large number.

When looking to achieve huge speed improvements to meet a growing class of application requirements, various vendors developed *Parallel Processors*, implying that the use of 100 processors would achieve speed multipliers approaching 100. Carefully tested results show that, except for applications known as "embarrassingly parallel," speed multipliers for most large applications are a small percentage of the number of processors, see [1]. Defining Processor Utilization Efficiency (PUE), this implies that 100 processors may be required to achieve a speed multiplier of 10 over the single processor speed, due to a PUE of 10%.

Two key properties of parallel processing are the *Independence* and *Understandability* of software processes. To run concurrently, two processes must be independent. *Spatial independence* implies that they share no data. *Temporal independence* implies that they may exchange data from time to time in a synchronized manner while running concurrently. Understandability implies that an application expert can understand the code representing the algorithms to ensure its correctness.

---

† Visual Software International - www.VisiSoft.com

Requirements For A New Design Direction

"Software gets slower faster than hardware gets faster"
Niklaus Wirth's law

Upon reading about the decline in software productivity since the early 1980s, it becomes clear that building software using C-based languages (C, C++, C#, Java, Python, etc.) and newer versions of Fortran have been a step backward - even on single processors. In the original designers' own words, C was never intended to be a real programming language. It certainly was not thought out as a platform for language development. All of the special language add-ons over the years have not solved the problem. Although OOP dominates current beliefs, there is no measure of productivity of this methodology. The decline in application speeds during the 1980s and 1990s led to Niklaus Wirth's law above.

As quoted below, top engineers describe the difficulty encountered using current software approaches, and the direction needed for change, see [6], [11], [14], and [22].

> **Justin Rattner** (former Chief Technical Officer, Intel),
> "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."
>
> **Chuck Moore** (formerly Chief Technology Officer, AMD)
> "The industry is in a little bit of a panic about how to program multi-core processors, especially heterogeneous ones. 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."
>
> **Craig Mundie** (Chief Research & Strategy Officer, Microsoft)
> "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 will be required, and could affect how almost every piece of software is written. --- This problem will be hard."

The above statements by top computer engineers imply that a new software technology is required for parallel processing. Even High Performance Computing (HPC) organizations are coming to that conclusion, see [13], [20]. Trying to see how many high-flop CPUs can fit into a warehouse is not the problem. The real requirement is for an approach that will *minimize the number of processors needed to meet the runtime constraints for a given application*. This implies a solution that achieves high PUEs, i.e., above 90%.

To achieve such a solution, one must go back to von Neumann's ISA and determine the equivalent for parallel processor design. Unlike Turing's machine, this is not a simple one-dimensional problem. Not only is it multi-dimensional in physical space, but the need to share information varies with time. Design of parallel processor architectures requires knowledge of the application requirements and particularly the solution spaces into which they must be mapped.

Application Space Architecture - The ISA For Parallel Processors

Parallel Processor Applications

Those who have worked many parallel processor applications, such as the ones enumerated below, know that each class of applications is different.

1. Wave Guide simulation
2. Human Body simulations
3. Global Climate prediction
4. Fluid Flow simulations
5. Biological Particle simulations
6. Chemical - Molecular structure simulations
7. Scanning, Sorting, and Correlating massive databases
8. Weather prediction - including mountainous terrain
9. Power distribution simulation
10. Electro-magnetic wave simulation
11. Global HF power transmission
12. Global military planning - Multiple moving platform simulations

Although most computer architectures being sold to this market are really server designs, these are not server applications. Server applications contain many tasks that can be run on different processors - because they are totally independent. Parallel applications are compute bound, not I/O bound as are server applications. They are generally single tasks with substantial inherent parallelism, where the parallel parts influence each other while they are running concurrently on separate processors (they are temporally independent).

Working Toward High Speed Parallel Processor Architectures

The major requirement in server applications is the need for communications relative to managing and running multiple independent tasks on a set of processors, and the continual allocation, tracking, and reallocation of memory segments to run them. The major requirement in parallel applications is the need to share memory directly among modules running concurrently on separate processors in a single task.

Parallel processor applications generally require front-end preparation and post-run analysis tasks that can be performed in a server environment. However, if the speed-bound tasks are not run on a well designed parallel processor, speed is reduced by orders of magnitude. Tasks that take a few minutes may take hours; those that take hours may take days. This huge discrepancy is removed by proper design of software as well as hardware architectures.

An example of separation of server and parallel processor hardware architectures is illustrated in Figure 1. The parallel processor environment has no direct I/O channels, but is interfaced directly using shared memory with the server for I/O.

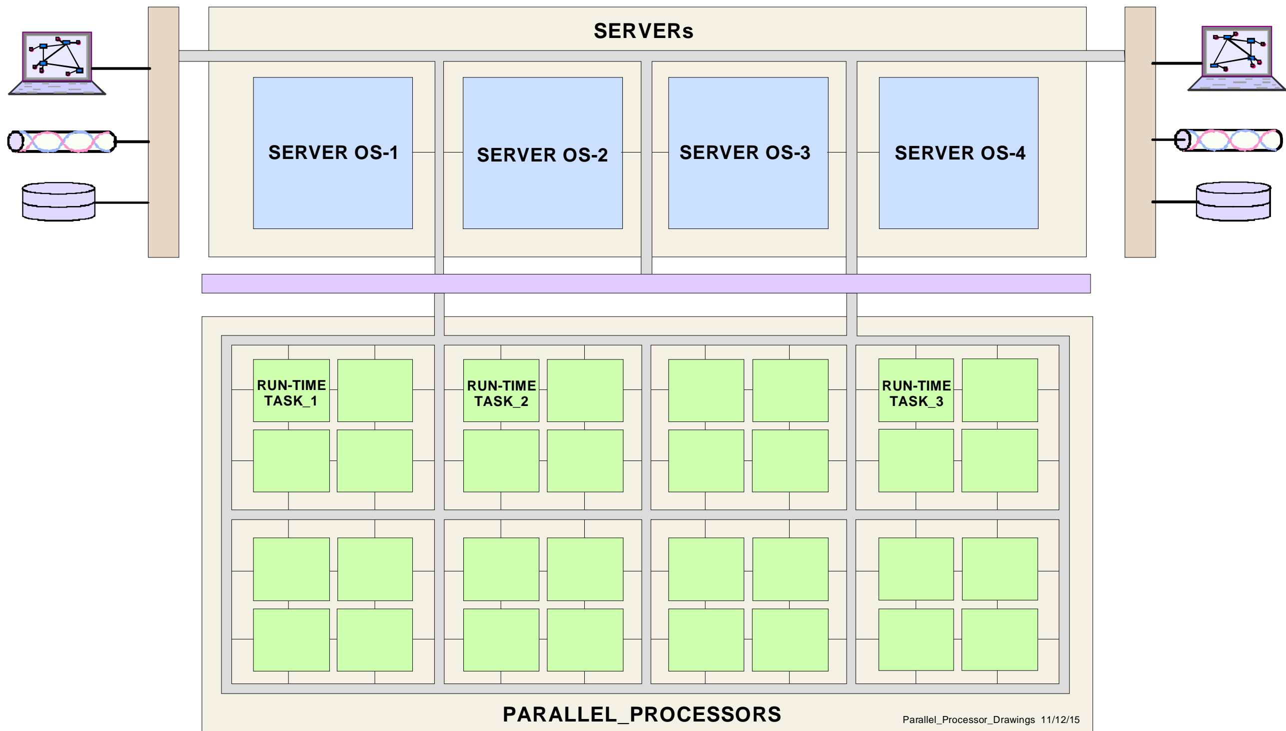

Figure 1. A parallel processor environment connected to a server environment.

Mapping Multi-Dimensional Application Spaces Into Hardware Spaces

The multi-dimensionality of parallel processing is apparent from Figure 1. Whereas one or more tasks in the server environment run on a single processor, tasks 1, 2, and 3 running on the parallel processor are using 8, 16, and 8 processors respectively. Unless a task is embarrassingly parallel, it must share memory between processors while running concurrently. The problem is to map the parallel processor application space for a task into a parallel processor space that maximizes run-time speed. This requires tailoring the parallel processor hardware space to the ASA. Initiated in 1982, the ASA defined here has evolved to support all of the parallel applications of which we are aware, and which are representative of all those looking for significant speed improvements.

For parallel applications, the ASA must support the spatial representations necessary to optimize mapping of the inherent parallelism in an application onto the resulting hardware architecture. It is the application requirements (for all of the parallel applications of interest) that define the best software environment to be used by the application experts. Using this software environment, application experts must be able to easily define the spaces that best support the desired operations and corresponding algorithms of their applications.

A CAD System That Represents The ASA

Instead of an ISA consisting of a set of assembler instructions, the environment for developing parallel processing software must provide application experts the ability to map the inherent parallelism in a system into a software architecture. This requires a combination of the following:

- Explicit definition of software modules that provide spatial and temporal independence;
- Visualization of the independence properties of the application architecture;
- A language that supports the required complex data spaces and architectural drawings;
- A language that supports understandability by application experts;
- A Run-Time System (RTS) that contains knowledge of the software architecture;
- An OS that takes full advantage of the RTS information;
- An OS that optimally maps parallel software architectures onto parallel hardware architectures;
- A Run-Time Environment that captures critical run-time statistics;
- Visualization of measures of PUE that helps designers maximize speed of operations.

Hardware architectures must be designed to support the Application Space Architectures generated by application experts using a tailored software environment, see Figures 2 & 3.

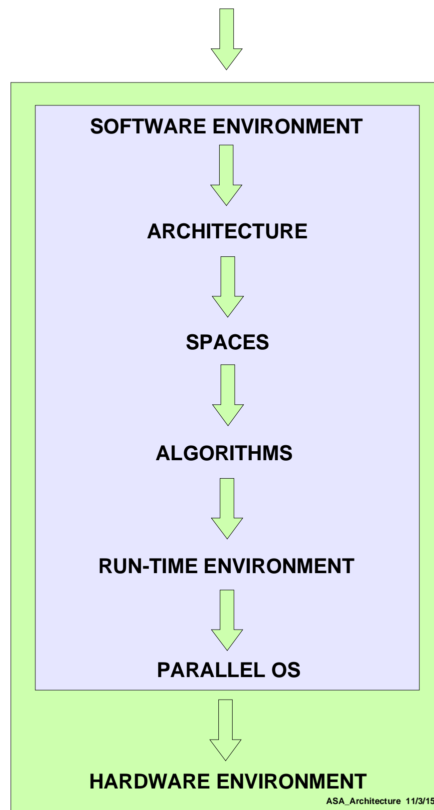

Figure 2. Illustrating development of the Application Space Architecture (ASA).

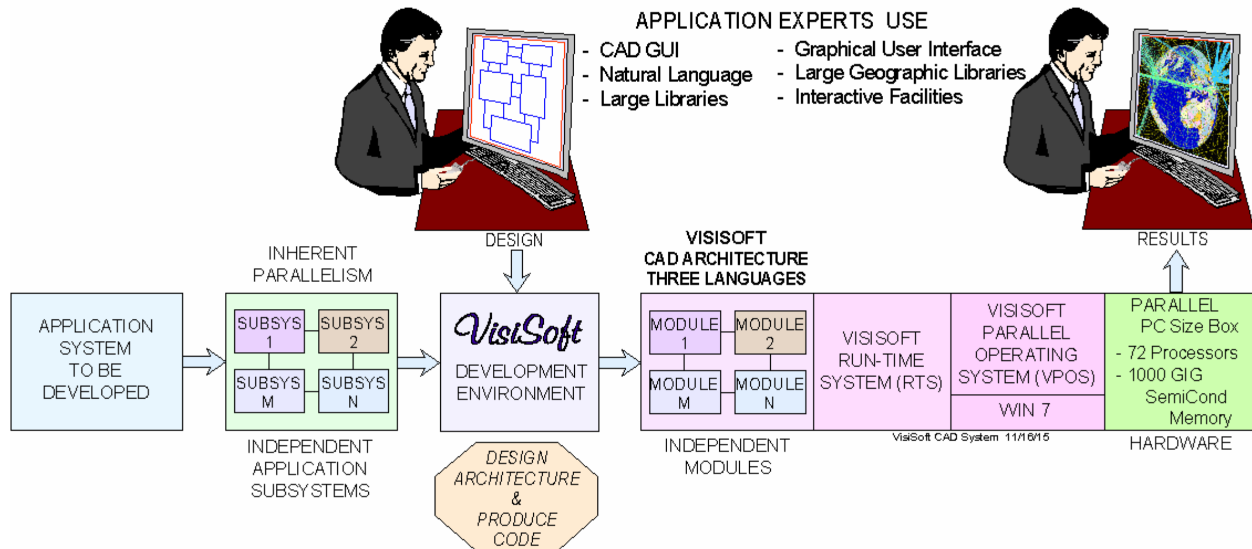

Figure 3. The ASA CAD development environment for parallel processor software.

The ASA solution described here is embodied in VisiSoft, a CAD system for building parallel processor software. Illustrated in Figure 3, it is well tested over many years, providing multiple order of magnitude increases in parallel processor speeds. Although the reasons for its success become obvious, development is based on many experiments using principles underlying multiple disciplines. These include mathematics, physics, engineering, information theory, communications, and discrete event simulation as well as software theory, large database design, and computer design. Substantial knowledge evolved from building and testing many complex applications such as those enumerated above.

Figure 4 illustrates the development of a software system. Applications may run on multiple processors with separate elements optimized to share local data memory and instruction memory. Subject area experts want to translate their problem directly into a user-friendly language. As shown by history, this language must be designed for human understanding, making it easy for experts to map their application into a software space that simplifies their problem.

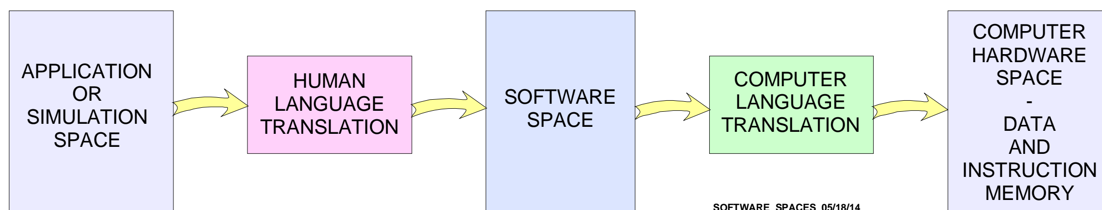

Figure 4. The environment of software language translators.

This implies simplicity of transformations and processor counts minimized while meeting run-time speed constraints. Translation of a human-oriented language into binary is done by the computer. In this CAD system, the human-oriented language is designed to simplify mapping applications into a tailored software space using engineering drawings for visualization. This puts the burden of complex translation on the computer where it belongs, see [21], pg 64.

Figure 3 is an expansion of Figure 4. It illustrates the translation of an application system composed of inherently parallel subsystems into a software system of Independent (IND) Modules that can run sequentially on a single processor or concurrently on different processors. When on the same processor, IND modules can share databases directly. When on separate processors the shared databases are automatically copied and synchronized by the Run-Time System (RTS) generated during the language translation process and shown in Figure 2. The VisiSoft Parallel OS (VPOS) uses information provided by the RTS and is tailored to parallel processing, as indicated in Figure 3.

Using the VisiSoft ASA (pink boxes in Figure 3) to define parallel processor requirements, hardware designs are simplified, maximizing memory close to the individual processors, and the speed of memory copies between processors, chips and boards.

Given languages that support the speed and architectural description of software modules, one must produce an RTS that contains information on the software architecture. The RTS is automatically generated along with the object code for the system. The RTS interfaces with the OS so that IND modules can be placed on processors in a manner that optimizes inter-module communications and run-time speed. This requires the VisiSoft Parallel OS (VPOS) which uses the RTS information to schedule and synchronize module operations. The RTS also provides VPOS the information necessary to map the overall software operation into computer hardware to maximize run-time speed.

Software Architecture

To implement software solutions for the applications enumerated above requires experts who understand the underlying physical computational requirements that determine the inherent parallelism in the application system. Given sufficient inherent parallelism, one must map that parallelism into a highly tailored software architecture. As described below, software architecture is embodied in both engineering drawings and languages.

Visualization of architecture is critical to software design where the spatial and temporal independence properties of the modules determine their ability to run concurrently on parallel processors. Visual representation of these properties must be displayed in a manner that simplifies understanding - by application experts. This requires specific properties of the language used to describe the data structures and algorithmic rules that define operational software. Without a language that supports these properties, speed cannot be achieved on single - as well as parallel - processors.

Hierarchies are key to both speed and control over complex software. This starts with syntactical facilities in the language for creating data and rule structures with deep hierarchies. Similar requirements exist for module structures, the RTS, VPOS, and finally the hardware - in that order. One must go up and down this chain - designing many complex applications - to understand the huge impact of hierarchical structures on speed and understandability.

Software Architectures For Parallel Processors

To optimize run-time speed, application designers must produce a software architecture that maps into the hardware architecture so as to minimize memory movement delays. To understand this, consider the following application properties.

> **Particle Motion** - Particles close to each other interact via gravitational forces. When a sufficient number of particles are close, those beyond a given distance have an insignificant effect on their movement. Particle interaction - and the need to share memory - is limited to those within a known distance. Those particles sufficiently close can be put on the same processor, or on processors that are physically close to each other. If the application is nonstationary, their connectivity will vary with time.
>
> **RF Communication Systems** - Communication system models follow physical network connectivity. Units that are connected can share data. In the case of radio systems, those within a given distance, and not blocked by terrain or other elements on the earth's surface, are connected. When they move, their connectivity will vary with time.

Having built many large scale simulations, including those for Big Data, only those like the radio network communication systems described above have nonstationary connectivity, making them far more difficult to design for a parallel processor.

Software Spaces & Databases - An Extension Of Mathematics

The CAD approach described here follows from Shannon's *Mathematical Theory Of Communications*, [18]. Based on binary numbers and Boolean algebra, it defines a space wherein the set of characters used to write software code is represented by strings of bits (binary numbers). This CAD approach also follows from the State Space framework formulated by control theory engineers to simplify complex control system design, [17]. State Space extends vector and matrix mathematics, simplifying complex transformations using large vector spaces.

Simplification of complex mathematical problems hinges on selection of a good space, reducing complexity of the transformation algorithms and the corresponding time to solve the problem. This becomes apparent when using VisiSoft to represent the multi-dimensional hierarchical spaces that occur in software. Software spaces are determined by the databases used to support transformations (instructions) that implement an application.

If a software application is represented by a continuous-time or discrete-time linear mathematical model, the software and mathematical solutions can be the same. However, most software applications require actions based upon events as they unfold, being highly nonlinear. Discrete event simulation provides significant insights into this problem, see [9]. Although time is still the basic coordinate, actions jump to the next scheduled event. The difference is that actions typically depend upon complex decision processes. e.g.,

```
IF A IS TRUE ... SCHEDULE PROCESS_A ... ELSE IF B IS TRUE ... SCHEDULE PROCESS_B
```

Some of the execution steps may involve solving systems of equations. They may also contain statements that SCHEDULE a NEW_EVENT in the future. This facility is necessary in discrete time models or real-time systems. Software is simply an extension of mathematics. The corresponding properties of spaces, and the independence of subspaces and coordinates, apply directly. These properties are critical to simplifying the design of software architectures for parallel processors and maximize speed.

Language And Information Theory

The more information one has to make a decision, the more likely a good outcome. If information is misunderstood, the probability of a poor outcome increases. Fast and reliable communications is the goal of Shannon's information theory, [18]. The basic principle is that: *Reliability of information transfers is increased by adding redundancy* (i.e., additional data). This principle is used to write and read computer memory. Code bits are added to the data being stored, decreasing the probability of error when reading it. In RF communications, one may double the size of the original data stream to ensure reliable transfer. One may send the same message twice, or use additional words, such as articles, adjectives, or adverbs.

English is considered to have a high degree of redundancy compared to other languages, increasing the probability of reliable communications and the survival of its users. As stated by Bjarne Stroustrup, creator 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," [19]. It dominates the world of free trade. Its success is attributed to its understandability.

Studies comparing interactive languages have shown that errors increase as statements move from good English to a more terse form, see [12]. Comparisons of COBOL, FORTRAN and C-based languages (C, C++, C#, Java, Python, etc.) will typically derive the following conclusions: COBOL is easily understood; FORTRAN is fair; C-based languages are terse, an objective of the principle designer to make the translator small and easy to write, see [16].

Generalized State Space

Generalized State Space capitalizes upon concepts of the State Space framework used in control theory by extending the mathematical definitions of vectors and transformations. Based on Shannon's theory of binary systems, we introduce 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 typical mathematical operators. For example, we may want to use English words as operators to say:

```
IF LIGHT IS YELLOW, SET LIGHT TO RED ,
```

a *Generalized Transformation*. Given this facility, one may view computer software as consisting of generalized state vectors (data) and generalized transformations (instructions). This framework for designing software is called *Generalized State Space*, see [3] and [4].

Hierarchical Data Spaces

Complex data structures are easily understood by application experts when put into their natural hierarchy. Figure 5 is taken from an embedded-software radio. Deep hierarchies allow large complex data structures (*Resources*) to be moved in a single instruction fetch, with all the individual fields directly available to instructions (*Processes*) that use them. This supports order of magnitude improvements in speed. Figure 6 is taken from a simulation used to design the radio. It illustrates an "Instanced" *Resource* supporting multiple transmitters and receivers. The QUANTITY clauses used to define tabular arrays in Figure 5 are eliminated. Multiple TRANSMITTERs and RECEIVERs are identified automatically by the *Processes* that use the instanced resource. This is because they are part of an *Instanced Module* defined below.

```
RESOURCE:  INDEXED_MESSAGE_TABLE

MESSAGE_TABLE     QUANTITY(3)
    1  MESSAGE_INDEX                                  INTEGER
    1  MESSAGE_ELEMENT   QUANTITY(13)
       2  UNIT_ID                                     INTEGER
       2  SLOT_ID                                     INTEGER
       2  MESSAGE_INFORMATION
          3  MESSAGE_TYPE                             STATUS  DATA_OUTPUT
                                                              USER_REQUEST
          3  STATE_S
             4  NUMBER_TO_BE_SENT                     INTEGER
             4  SEQUENCE_NUMBER                       INTEGER
             4  MESSAGE_ACTION                        STATUS  SEND, HOLD
             4  AGGREGATE_STATE
                5  MESSAGE_STATE  QUANTITY(7)         STATUS  EMPTY, FULL
             4  INDIVIDUAL_STATE  REDEFINES AGGREGATE_STATE
                5  SEQUENCED_MESSAGE
                   6  GROUP_MESSAGE                   STATUS  EMPTY, FULL
                   6  BUDDY_MESSAGE                   STATUS  EMPTY, FULL
                   6  QUEUED_MESSAGE                  STATUS  EMPTY, FULL
                   6  RESERVED_MESSAGE                STATUS  EMPTY, FULL
                   6  INTERCOM_MESSAGE                STATUS  EMPTY, FULL
                5  NON SEQUENCED_COMMAND
                   6  DATA_INPUT                      STATUS  EMPTY, FULL
                   6  USER_COMMAND                    STATUS  EMPTY, FULL
```

Figure 5. Example of a hierarchically structured Resource.

```
RESOURCE:  TRANSCEIVER                       INSTANCES:  TRANSMITTER
                                                         RECEIVER

GENERAL_PARAMETERS
    1  TRANSMITTER_POWER             REAL   INITIAL_VALUE 100
    1  RECEIVER_THRESHOLD            REAL   INITIAL_VALUE 120

RADIO
    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
    1  POWER_AT_RECEIVER             REAL
    1  TOTAL_NOISE_POWER             REAL
    1  CONNECTIVITY_MATRIX
       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 6. Building hierarchical data structures using an Instanced Resource.

In the resource in Figure 6, instances are automatically set at the module level when a process within an instanced module is CALLed or SCHEDULEd. Moving the instance implementation to the architectural level corresponds to the design of physical systems, substantially improving understandability of the code by application experts.

Hierarchical Transformations

Figure 7 is a VisiSoft *Process*. The hierarchical organization applies directly to simplification of complex instruction sets using one-in one-out control structures, see [15]. With automatic instancing, subscripts corresponding to quantity clauses in Figure 5 are unnecessary.

```
PROCESS: RECEPTION                    RESOURCES: TRANSCEIVER
INSTANCES: TRANSMITTER                           MESSAGE_FORMATS
           RECEIVER                              TRANSMITTER_OUTPUT

START_RECEPTION
    IF TRANSCEIVER IS IDLE
        EXECUTE GOOD_RECEPTION
    ELSE IF TRANSCEIVER IS RECEIVING
        EXECUTE CONFLICTING_RECEPTION
    ELSE IF TRANSCEIVER IS TRANSMITTING
        EXECUTE CONFLICTING_BROADCAST .

GOOD_RECEPTION
    IF SIGNAL_TO_NOISE_RATIO IS GREATER THAN RECEIVER_THRESHOLD
        SET TRANSCEIVER  TO RECEIVING
        ADD SIGNAL_POWER TO TOTAL POWER_AT_RECEIVER .
        CALL DECODE_MESSAGE .

    IF  MESSAGE TYPE IS FORMAT_A
    AND SYNC_CODE    IS VALID
    AND LAST_SYMBOL  IS A TERMINATOR
         EXECUTE SEND_ACKNOWLEDGEMENT .

CONFLICTING_RECEPTION
    IF POWER_AT_RECEIVER IS GREATER THAN SIGNAL_POWER
         SCHEDULE ABORT_RECEIVE NOW .

CONFLICTING_BROADCAST
    CANCEL END_RECEIVE NOW
    SCHEDULE START_RECEIVE IN EXPON(0.83) MILLISECONDS
        WITH PRIORITY 80

SEND_ACKNOWLEDGEMENT
    MOVE ACKNOWLEDGEMENT TO TRANSMIT_MESSAGE_BUFFER
    IF DESTINATION IS BROADCAST
        SEARCH LINK_CONNECTIVITY_VECTOR OVER RECEIVER
             EXECUTING TRANSMISSION
                  WHEN LINK IS GOOD
    ELSE EXECUTE TRANSMISSION .

TRANSMISSION
    SCHEDULE LINK_RECEPTION
        IN LINK_DELAY MICROSECONDS
            USING TRANSMITTER, RECEIVER
```

Figure 7. Building hierarchical rule structures using the Process Language.

As described by Shannon, [18], the language used to transfer information plays a major role in ensuring its correct reception. As shown by Ledgard, [12], languages used to specify complex algorithms play a major role in the ability to understand software. The English language is known for its redundancy, a major factor in transferring understanding. As declared by Grace Hopper, world-wide expert in software language design, software languages must be easy to understand - and read like English, the accepted international language, see [2].

When building complex software, human translation is simplified if a language supports obvious representation of physical behavior. The examples in Figures 5, 6 and 7 are taken directly from embedded software or simulations of complex radio communication systems. With hierarchical data structures like those shown, one can represent the complex algorithms associated with physical systems with ease. This is illustrated in Figure 7. These systems entail more complex resources and processes than those shown, but all are easily understood by application experts, the only ones that can ensure correctness of the algorithms.

Engineers concerned with modeling and simulation of complex systems can easily represent their problem in these understandable languages. Those who review the work of the developers can easily understand the spaces and rule hierarchies to verify and validate the design. Although the language looks "verbose" to programmers, they are surprised at the speed with which VisiSoft software runs - typically 10 to 100 times faster than the same application built in a C-based language or FORTRAN - on a single processor, see Chapter 17 in [5].

The Separation Principle

The underlying principle supporting the visualization of software architectures using engineering drawings is the separation of data from instructions at the language level. Defined in 1982 in the design of the General Simulation System (GSS), [9], this has become known as the *Separation Principle*, [10]. The developers of GSS defined the separate languages used to describe the data structures (*Resources*) and rule structures (*Processes*) illustrated above.

Using the Generalized State Space framework, the Separation Principle is achieved by storing all data in *Resources*. Resources are depicted as ovals in architectural drawings as illustrated in Figure 8. *Processes* containing instructions that implement transformations are depicted as rectangles. The lines connecting them determine which processes have access to what resources. In this figure, each process has a dedicated resource and shared resources. 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.

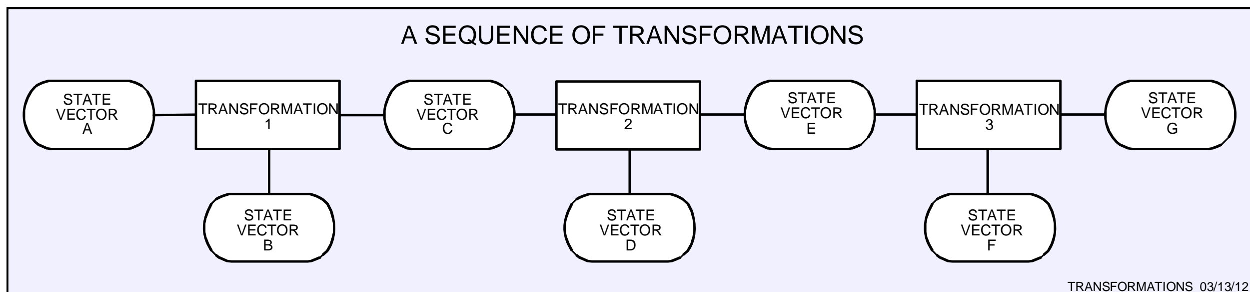

Figure 8. State vectors and transformations.

Consider that state vectors C, D, and E have initial values Ci, Di, and Ei. When run on a single processor (sequential machine), Transformation 2 will produce the same outputs: Co, Do, and Eo 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 *spatially independent*. Transformation 2 can operate only when Transformations 1 and 3 are both idle, i.e., they are *temporally independent.*

The *Separation Principle* provides the ability to represent resources and processes using icons on engineering drawings of software, see Figure 9. Engineering drawings represent the *connectivity* of elements; they are not flow charts. They provide an iconic visualization of which processes share what resources, and therefore their independence. All resources are shared by pointer. There is no global data! By grouping icons into hierarchies of modules, module independence can be visualized directly. Figure 9 is a Library type module.

Modularity & Independence

In engineering, breaking complex systems into independent modules is embodied in the architecture, a concept 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*. 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, as shown in Figure 9. Only then can one visually observe independence - the key property supporting concurrency on parallel processors. Flow charts, or graphical variations of flow charts, are of little use when describing the property of independence.

Figure 9 illustrates a decomposition following the organizational lines of the application (predicting electro-magnetic wave propagation in mountainous terrain). The architectural decomposition, and design of the data spaces, individual modules, and algorithms all require substantial application expertise. The interface to this library is a single resource comprised of a large hierarchy of elements that runs about 2 pages and is moved with a single instruction fetch.

In VisiSoft, which processes have access to what resources is determined solely by the architecture - not the code; the languages do not permit declaration of scope rules. This includes the manner in which data is shared - simplex or duplex, and who has control. Different colors denote different resource and module types. Most important, the languages are designed to provide for deep hierarchies in both data and rule structures yielding greatly simplified architectures. Without these language properties, the ability to understand complex software decreases exponentially. The value of these properties is most apparent when building complex systems, and especially for parallel processors where modules must be independent to run concurrently on separate processors.

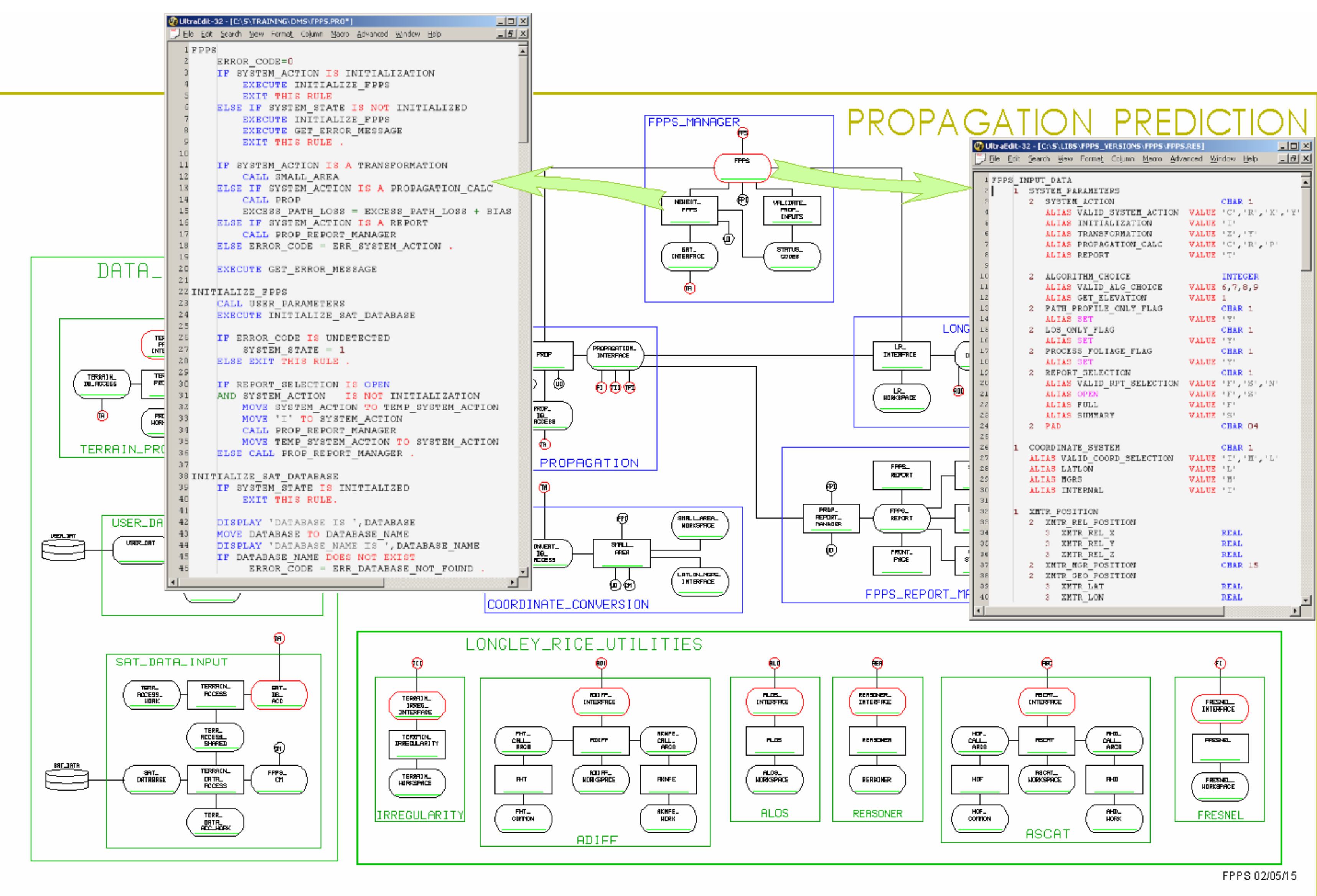

Figure 9. Illustration of editing resources and processes on the engineering drawing.

Visualization

Engineering fields (e.g., Aeronautical, Electrical, Mechanical, etc.) would be at a huge disadvantage without engineering drawings. As system complexity increases, it is necessary to create hierarchies of modules, where the number of levels in the hierarchy is determined by system complexity. The higher the complexity, the deeper the layers of hierarchy required. This determines the understandability of the overall system and time to develop and enhance a reliable product.

Without drawings of software, sharing of data is often driven by the desire to save memory and share variables. Dependent interconnections become common, and modification of one part of a system affects other parts thought to be independent, see [7]. In Figure 9, modules may be replaced without changes to the rest of the system. Using copies of memory, movement is minimized, improving speed while substantially simplifying the use of parallel processors, and increasing Processor Utilization Efficiencies (PUEs) to between 85% and 95%.

Parallelism, Architecture, And Decomposition

Parallel Processor Architecture Requirements

As stated above, one must be able to map the inherent parallelism in an application into a software architecture such that IND modules can run concurrently. This implies creating IND modules that only share data using Inter-Processor (IP) resources. To determine the independence of IND modules, designers must easily see which processes share what IP resources. This is only done when the following critical requirements are met:

- Resources are organized into a minimum number of large hierarchical structures to represent the best spaces to implement problem solutions;
- Designers can visually determine from the drawings which processes share what resources to assure module independence.

Discrete Event Models

The speed of many applications, including weather prediction and military planning, can be improved by orders of magnitude using a discrete event approach. (Discrete event approaches originated in commercial banking and industrial planning.)

Using a discrete event model, the time to schedule a future process can be determined when a particular action is taken. Scheduling processes at the time of the action eliminates the need - at a future time - to loop through a sequence checking values, or to check a list to determine if a flag has been set indicating the need to invoke a process. Each of these actions wastes considerable time. Using discrete event models, the process is invoked when popped at its scheduled time, a facility provided by VPOS.

Design of the language, and the supporting scheduling and queuing software, is critical to the speed of a discrete event approach. Using the CAD system described here, enormous amounts of time are saved (otherwise wasted looping and testing), compared to other discrete event approaches. Processes are scheduled for a specific time (can be generated randomly) with the ability to cancel their future occurrence. The event scheduler is a critical part of VPOS.

Software Languages To Support Parallelism

Resource and process language requirements were driven by factors similar to those for tiling in parallel versions of FORTRAN. Using VisiSoft resources to interconnect modules, one can also interface temporally independent modules that use different coordinate systems. These factors eliminate memory management overhead for swapping and paging. PUE is raised by eliminating the overhead on each processor.

To do this, languages must support design of software spaces that simplify the human translation of inherently parallel physical entities into an organization of independent workloads. As understood by Grace Hopper, [2], such organizations are best supported by deep hierarchies of both data and instructions.

When the instruction language supports large hierarchies of rules, both looping and complex IF ... THEN ... ELSE statements are flattened. What is known as *Waterfall* or *Fall through* code is gone (without GOTOs). These properties dramatically simplify the design of complex algorithms, leading to substantial increases in both understanding and run time speed - on single as well as parallel processors, see [4] and [5].

Inter-Processor (IP) Communications

IND Modules residing on separate processors must communicate with each other. Design of the resources used to communicate between them must ensure their temporal independence and ease of understanding of the module designs. Results obtained within one IND module are typically communicated in one direction for use by others. This is convenient since, if IND modules are exchanging data, exchanges must be synchronized to run concurrently. Sending and receiving data efficiently time-wise is best done using separate simplex channels, allowing synchronization to be performed easily.

Using the VisiSoft CAD system, architectural design for communication between IND modules residing on different processors is accomplished using Inter-Processor (IP) resources. These provide a simplex channel written by only one IND module, and only read by other IND modules. Two-way communication is accomplished using a pair. Synchronization of IP resources is done automatically using copies in the RTS.

Parallel Processor Architecture Requirements

Mapping the inherent parallelism in an application into a software architecture such that IND modules can run concurrently implies creating modules that only share data using IP resources. To determine the independence of modules, designers must easily see which processes share what IP resources. This is only possible when the following critical requirements are met:

- Resources can be designed with deep hierarchies to represent the best spaces, where best implies the simplest and fastest algorithms;
- Resources are organized into a minimum number of large hierarchical structures shared between processes;
- Designers can visually determine from the drawings which processes share what resources and how they are shared to assure module independence.

Taking Advantage Of Architectural Information At Run-Time

To take advantage of a parallel processor at run time, the OS typically maps threads onto processors to maximize the speed multiplier. A designer faced with generating complex algorithms should not be concerned with this problem. Similarly, traditional compilers will have little success trying to interpret an architect's decomposition of modules based on the code - specifically their independence properties and how they are synchronized to provide temporal independence. Finally, the operating system will not be very successful in determining where to map threads based on current run-time statistics, especially when they are nonstationary.

Underlying the VisiSoft CAD system are three language translators as indicated in Figure 10: one for data structures (resources); one for rule structures (processes); and one for run-time control (control specifications). Control Specifications render the software independent of the OS and platform, eliminating scripts while supporting the specification of complex databases, graphics, and allocation and synchronization of modules on parallel processors. When the Control Specification is translated, the Run-Time System (RTS) is generated - based on the architecture - to optimize the placement and synchronization of IND modules.

To make it easy for humans to understand, the languages are all context oriented, requiring the three computer language translators to be extremely complex pieces of software. Figure 10 illustrates a large simulation used for military planning, containing many types of fast moving platforms communicating within and between multiple IND modules. Numerous experiments have been conducted using this system on a parallel processor. Many of these are described in [5].

Temporal Independence And Synchronization

When processes are connected to the same resource but inhibited from running concurrently through synchronization on a parallel processor, they have *temporal independence*. Synchronization is automatically invoked by VPOS using information provided by the RTS. As stated above, IND modules communicate using IP resources. Two-way communication is implemented using two IP resources - one in each IND module that writes to that IP resource, with copies to all that must read it.

IND modules are automatically synchronized in time within a user-specified ΔT *Time Interval* by VPOS to ensure that the results are complete and consistent, see [5]. ΔT is defined by the designer in the Control Specification based upon accuracy requirements on the application results. By *synchronizing the release of, and access to copies of IP resources*, the temporal independence of IND modules that communicate with each other is automatically ensured. This architectural facility allows IND modules that share IP resources to run concurrently.

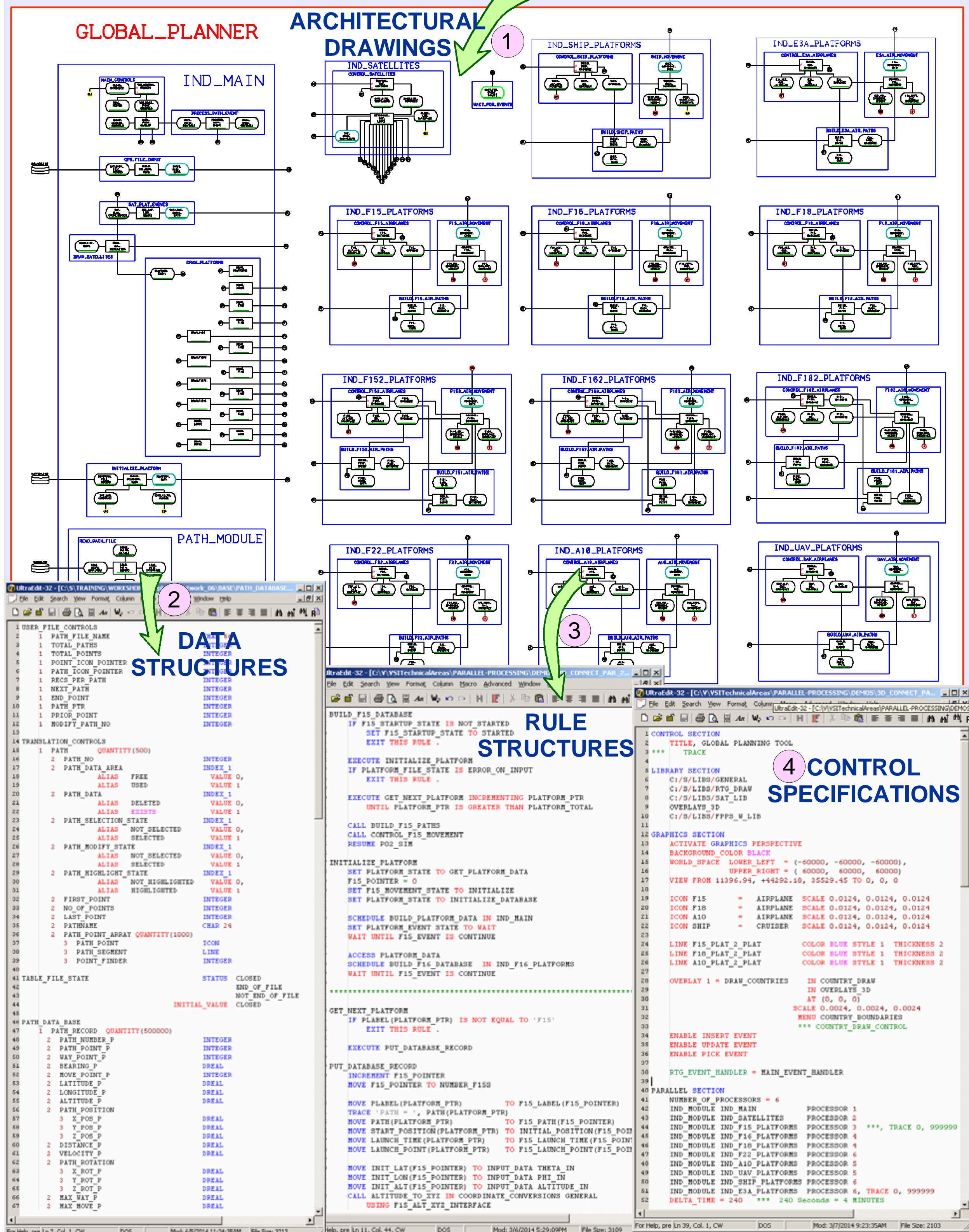

Figure 10. Example of an engineering drawing of parallel processor software.

The VisiSoft Run-Time System

The architectural information that characterizes the inherent parallelism of an application system is contained in databases that support the CAD development environment. The RTS is generated from that information to control VPOS calls that assign modules to processors and synchronize the use of IP resources. It also ensures that the resources reside with the processes that use them.

VisiSoft IND modules are typically large and remain on a specified processor, minimizing if not eliminating swapping and paging, and substantially increasing Processor Utilization Efficiency (PUE). However, as processor loads become unbalanced, PUE may fall and one must carefully consider the application level design constraints, typically to minimize the number of processors required to meet a given run time constraint. The VisiSoft CAD system contains built-in measurements and graphical depictions of the PUE, by IND module, so that users can easily assess and balance the loading on each processor, see Chapter 18 in [5]. By combining many IND Modules on a single processor, Processor Utilization Efficiencies (PUEs) are typically 85% to 95% versus HPC PUEs of 7% to 10%.

Parallel Processor Hardware Design Considerations

Hardware designs must focus on supporting the speed of operations of a large number of parallel processors packed tightly together to minimize propagation delays between individual processors and their data and instruction memory, as well as the memory the processors share. Going back to the von Neumann architecture, memory was critical to speed. Fortunately today, it is abundant. Chip space is best used by putting maximum memory close to the processors and eliminating functions that are not necessary for speed. Parallel processors may then be tied to server systems via special communication channels as shown in Figure 1.

This approach separates the functions requiring different channels to the outside world, allowing both the hardware and OS level design to focus on speed of operations internal to a single application. It also permits those functions that must interface with external communication channels, mass memory, and human interfaces to do so using full duplex channels composed of dual independent simplex channels that are easily synchronized. It also provides for multiple tasks running independently - concurrently - in the parallel processor. Scheduling, initiation, and termination of tasks can be managed from the server environment, while resource management, scheduling and synchronization of IND modules is done by VPOS on the parallel processors.

A less obvious difference is dealing with delays between processors that vary with the number of interconnections as well as distance. These delays can vary with the state of the application system itself, specifically the potential nonstationary connectivity of its IND modules. The delays result from the physical distance between individual cache and RAM memory segments serving large numbers of processors. When moving blocks of memory from chips to boards to trays to racks, delays become a major factor affecting application speed.

Memory Boundary Crossing Delays

To illustrate the effects of hardware architecture, Table 1 provides examples of possible ranges of time delays of 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 the processor (level 1 cache) or chip (level 2 cache) may take 1 to 5 nanoseconds. This may rise by a factor of 2 to 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 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 the architecture. We also note that these numbers have been improving with time and may continue to do so for the foreseeable future.

Table 1. Processor and memory - sizes and delays.

DELAYS, MEMORY SIZE, & PROCESSOR COUNT				
Position	Time Delays (Nanosecs)	Memory Type	Memory Size (Bytes)	Number of Processors
Within 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
Between Racks	1000 - 5000	SSD	1 - 100 P	32,000 - 1,600,000

Memory Maps 11/19/15

The key factors in the table are time delays (in nanoseconds), not transfer rates or frequencies (as in Gigabytes/second). They are critical in determining the run-time speed for most memory transfers - they are typically less than 1MegaByte. Frequencies remain constant over long distances and can achieve Gigabyte per second speeds. They can affect delays when sending data blocks in excess of 1 Megabyte. The above delays directly affect speed independent of the size of the transfer, and they do not remain constant. Delays tend to increase exponentially with distance, due to circuit designs that are required to maintain a recognizable signal (Shannon's theory). When going beyond the board, one is faced with delay multipliers on the order of 20 or more compared to level 1 cache.

Figure 11 provides a simplified illustration of the range of differences in memory boundary crossing delays for 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 starts to depend upon distance as well as the local hardware itself. The numbers used are only intended to show the desire to remain within a given hardware architectural footprint. 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 this sharing, minimizing the off diagonal transfers.

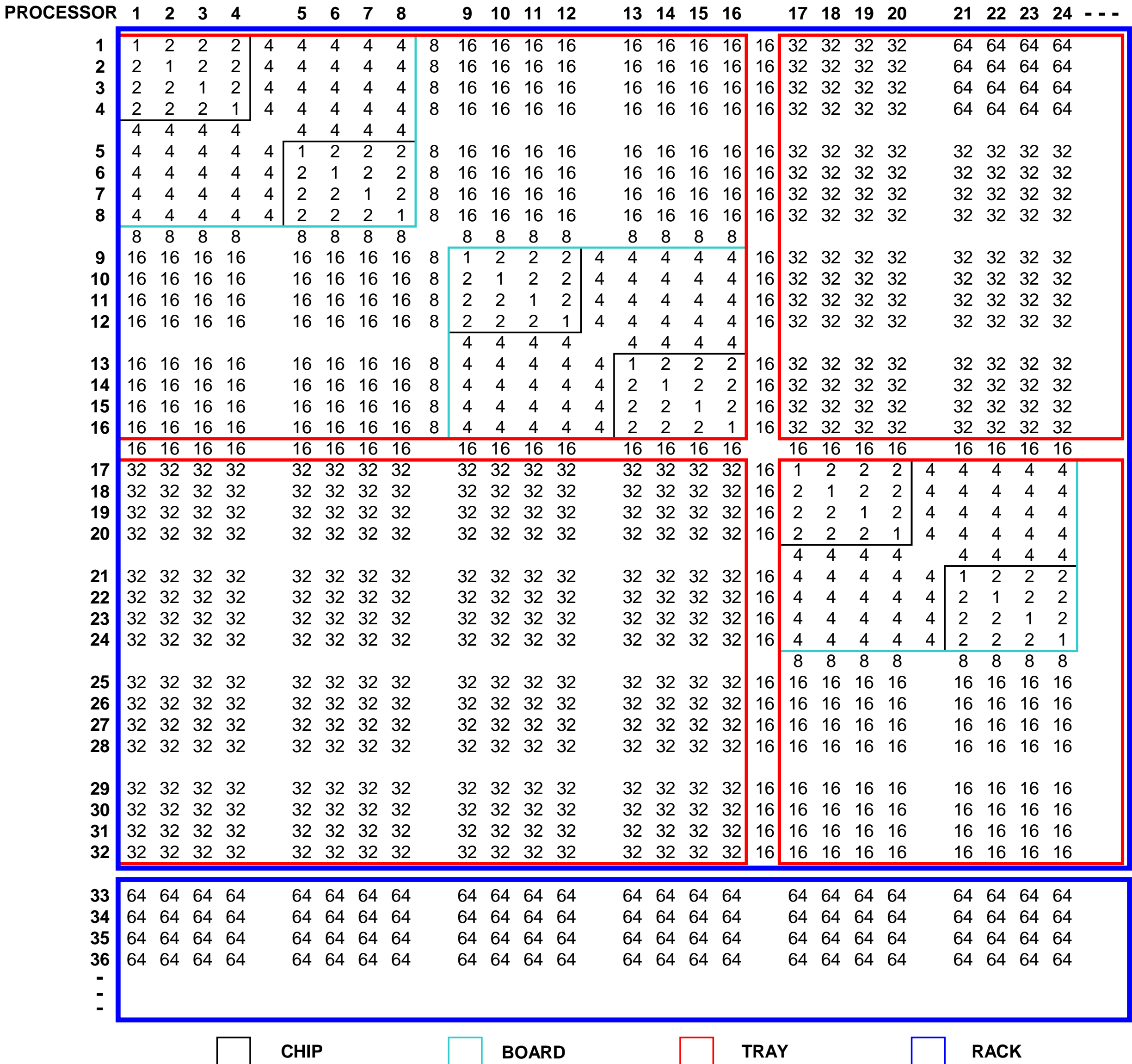

Figure 11. A simplified illustration of memory boundary crossing delays.

From Figure 11, it is possible that, using a good software architecture on a 32 processor PC board, one 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. Past experiments have shown that adding more processors can cause substantial reductions in PUE, and the corresponding speed multipliers one may expect from a large increase in the number of processors used for an application.

Mapping Software Architectures Onto Hardware Architectures

The architecture of the TELEPHONE_NETWORK Model is shown in Figure 12. This architecture contains 6 elementary modules and 1 hierarchical module. Figure 13 provides a connectivity matrix of the architecture that has already been diagonalized.

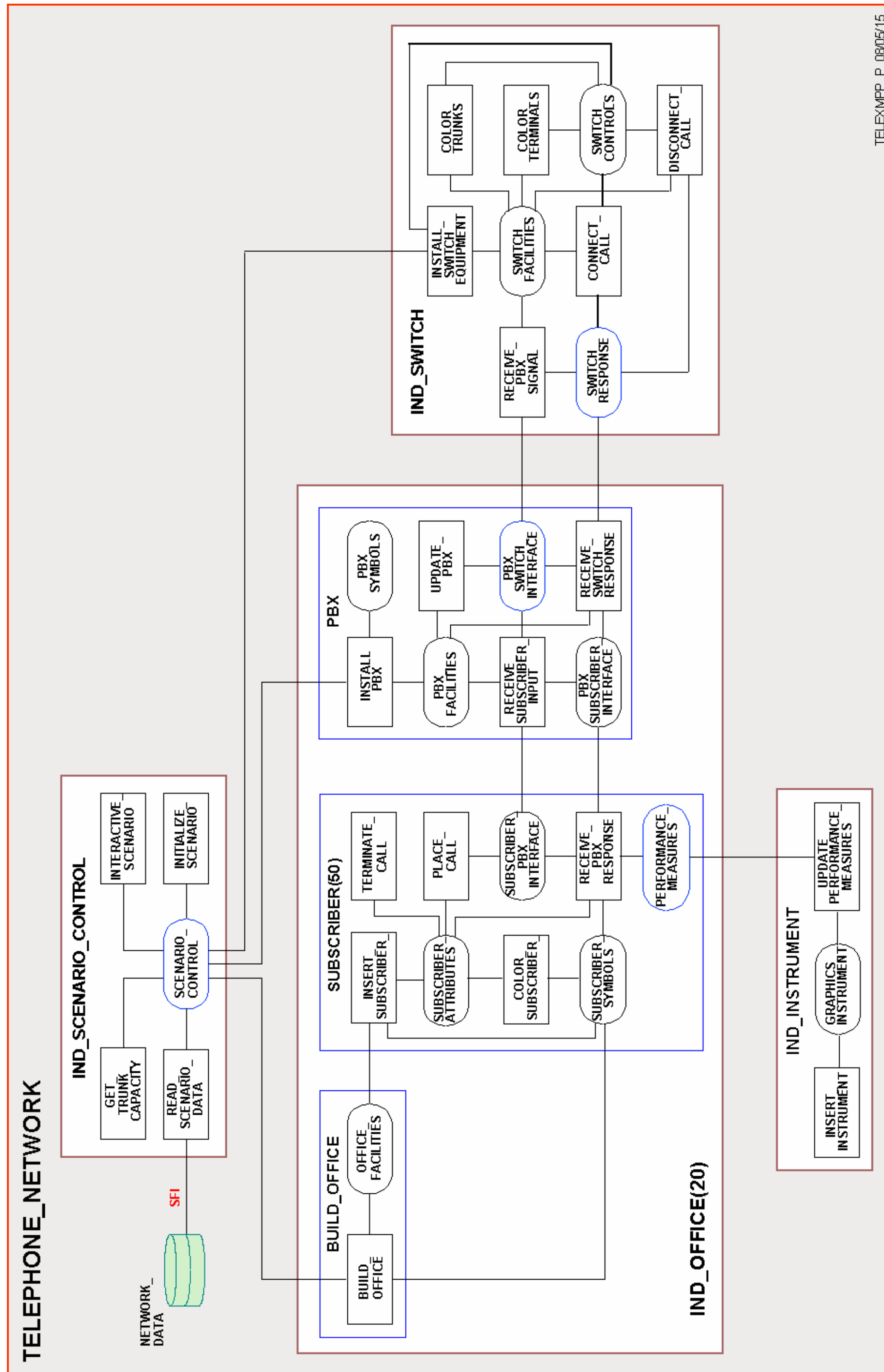

Figure 12. TELEPHOE_NETWORK Model.

In the matrix in Figure 13, the Xs indicate shared resources that are both read and written by processes in the modules shown. The Rs indicate resources that are only read by the marked processes. The main modules, SCENARIO_CONTROL, OFFICE, SWITCH, and INSTRUMENT, can be IND Modules since the resources shared between these modules are only read by processes outside the modules that write to them.

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	IND MODULES
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 11/19/15

Figure 13. Illustration of a diagonalized connectivity matrix.

Although this is a simple model, it illustrates the natural ability to create IND Modules of physical systems. Most important, instead of diagonalizing a matrix of numbers, we are diagonalizing a matrix of IND modules to minimize boundary delays on a parallel processor.

The above results illustrate a definition of embarrassingly parallel where the degree of embarrassment is determined by the diagonalization property. If the matrix can be diagonalized so no IP resources exist between IND modules, the application is embarrassingly parallel and can be run on a cluster. An example is Monte Carlo simulation where each simulation starts with a different random number seed. That’s certainly not the case with the applications of concern here, but a good architecture can take advantage of the temporal independence.

Summary Of ASA Software Facilities

Mapping software architectures onto hardware to minimize the run time for a given application requires much more than visualization of the software architecture. VisiSoft automates the mechanisms that ensure synchronization and protection of resources shared between processors. Specifically, designers can use the following architectural facilities:

- Resources can be organized into a minimum number of large hierarchical structures to represent the best spaces to implement fast problem solutions.
- Designers can visually determine from the architectural drawings which processes share what resources and how they are shared to ensure module independence.
- Designers can create IND Modules that run on a single processor.
- IND Modules share data using IP Resources that are automatically synchronized to ensure temporal independence. They can run concurrently with other IND Modules.
- Processes within IND Modules can write to IP Resources contained within the same IND Module, and release copies to other IND Modules.
- Processes within IND Modules can access and read the latest copy of IP Resources released by their controlling IND Modules.
- Threads run sequentially (they start with a process that can contain CALLs to other processes that are part of that thread).
- The process that starts a thread is SCHEDULEd by one or more processes that may also be part of that thread.
- Threads are contained within a single IND Module.
- IND Modules may contain multiple threads.
- Threads in IND Modules can SCHEDULE threads in other IND Modules.

Global Planner Example

The architecture shown in Figure 14 contains 16 IND Modules, one for satellites, one for each of 14 additional platform types, and one for initialization and graphical output. There are a total of 153 platforms on 30 paths. There are 17 different IP Resources (green borders): one for platform initialization, one for path initialization, and one for each platform's movement. There is only one case where an IND Module (satellites) has access to all other IP Resources. Since this simulation is running on a single board with 16 processors, speed is hardly affected by these memory accesses. When spread across a large number of processes, this problem is easily solved by splitting or copying this model to the other processors.

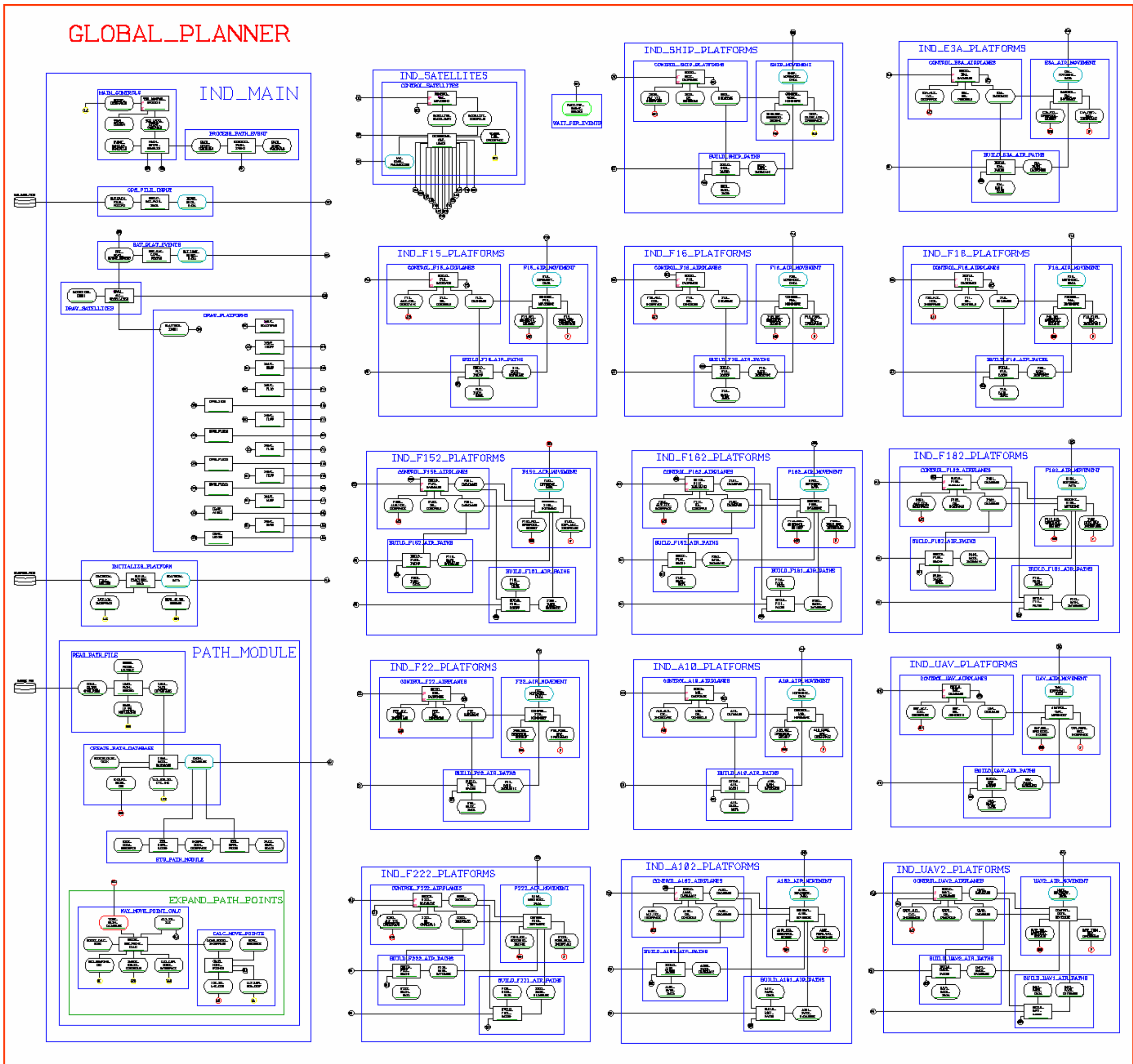

Figure 14. Illustration of IND models in a parallel processor simulation.

FAIR COMPARISONS BASED UPON EXPERIMENT AND MEASUREMENT

When conducting experiments to compare software languages and development environments, one must avoid the pitfalls that produce biased results, see Bailey, [1]. One is the wide range of software applications, from personal to commercial, to industrial, to government and military. The last two categories have invested the largest share of money in parallel processing, but are not as concerned about economics as a private business. When driven by economics, one wants to minimize the number of processors required to meet the speed constraint of a given application.

When making comparisons of parallel processor speed multipliers, one must use the same (fastest) single processor speed. Figure 15 shows the results of experiments manipulating a large database where different spaces (data structures) were used to implement the design, see Chapter 17 in [5]. The improvements in speed are due to increasing the depth of hierarchies, achieving a multiplier of 400. If comparison A uses test 1as the single processor test, the parallel processor speed multiplier will be 400 times faster than comparison B using test 6 for the single processor test. This points out the concerns of Bailey, [1]. Alternatively, using the approach in test 6, one may cut the number of processors by 400 to achieve the speed multiplier from test 1.

As stated above, using the VisiSoft environment, one is likely to obtain a PUE between 85% and 95%, compared to the 7% to 10% typically achieved by current HPC approaches, see Chapter 18 in [5]. Coupling this with other improvements easily provides another factor of 10, a direct multiplier on speed. Adding to this the single processor speed gains of 10 to 100, one can gain an overall factor of 100 to 1000.

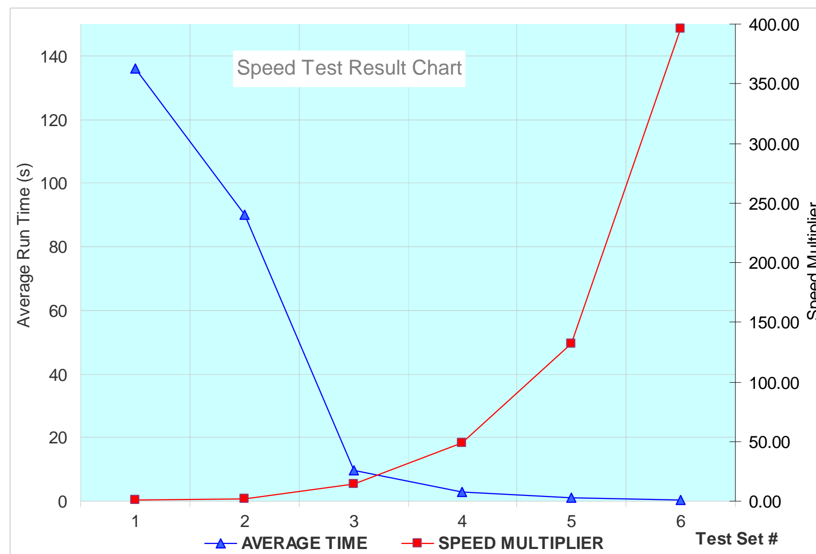

Figure 15. Single processor speed multipliers using different design spaces.

A 36 processor PC can be used to meet a speed constraint requiring a 3600 processor HPC. Just the reduction of distance between processors can gain another factor of 10 increase in speed. So combining these factors, an overall speed multiplier of 10,000 may be achieved. Testing shows that the speed multipliers using VisiSoft on a PC will fall within the range of 100 to 10,000 over an HPC depending upon the application and the current software approaches. Experimental testing by independent parties to prove these claims is welcome.

Why Application Experts Can Easily Design Parallel Processor Software

Only experts in an application understand the inherent parallelism within their applications. They know what modules can operate in parallel. As modules operate concurrently, they exchange information that affects their future behavior. In general, this is not a many to many exchange, but a one to adjacent exchange, e.g., from the center box to 26 adjacent neighbor boxes illustrated in Figure 16. Using 27 boxes, the center box can directly affect all of its 26 neighbor boxes supporting a 3D spatial environment as illustrated in Figure 17.

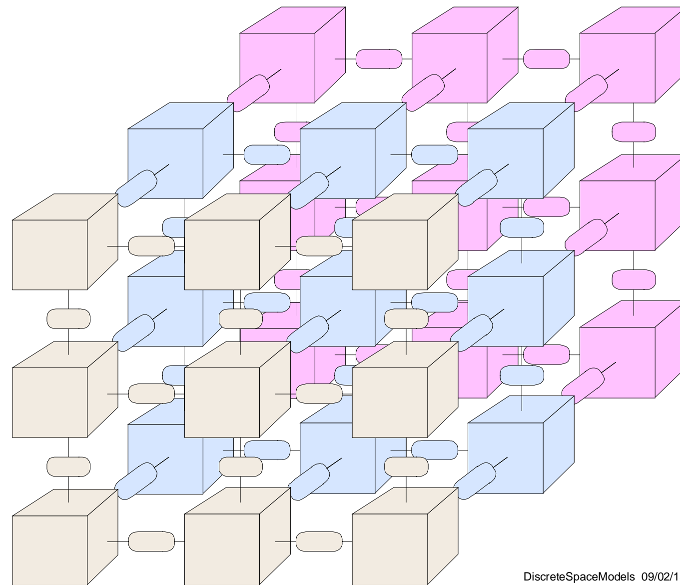

Figure 16. Example of 27 interconnected boxes

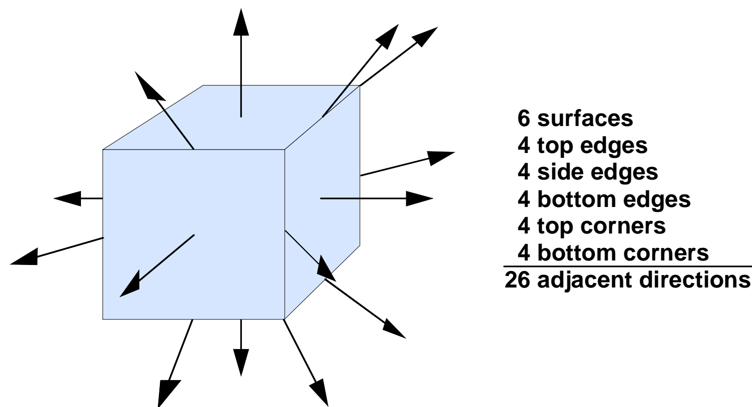

Figure 17. Adjacent surface, edge, and corner connections of boxes.

Figure 18 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 to eliminate sine and cosine calculations as waves travel through space in straight lines.

The space in which to map global applications, shown in Figure 16, is not immediately apparent - except to application experts. Physical operations are affected by coordinates representing the surface of the earth and the space above it. The earth's surface maps close to a sphere with (R, Θ, Φ) coordinates. This space maps into the hardware space shown in Figure 16. For applications requiring substantial GPS accuracy, Θ is the Latitude coordinate running from (-90° to +90°) corresponding to boxes running from bottom to top in the drawing. Φ is the Longitude coordinate running from (-180° to +180°) corresponding to boxes running from left to right and around the back.

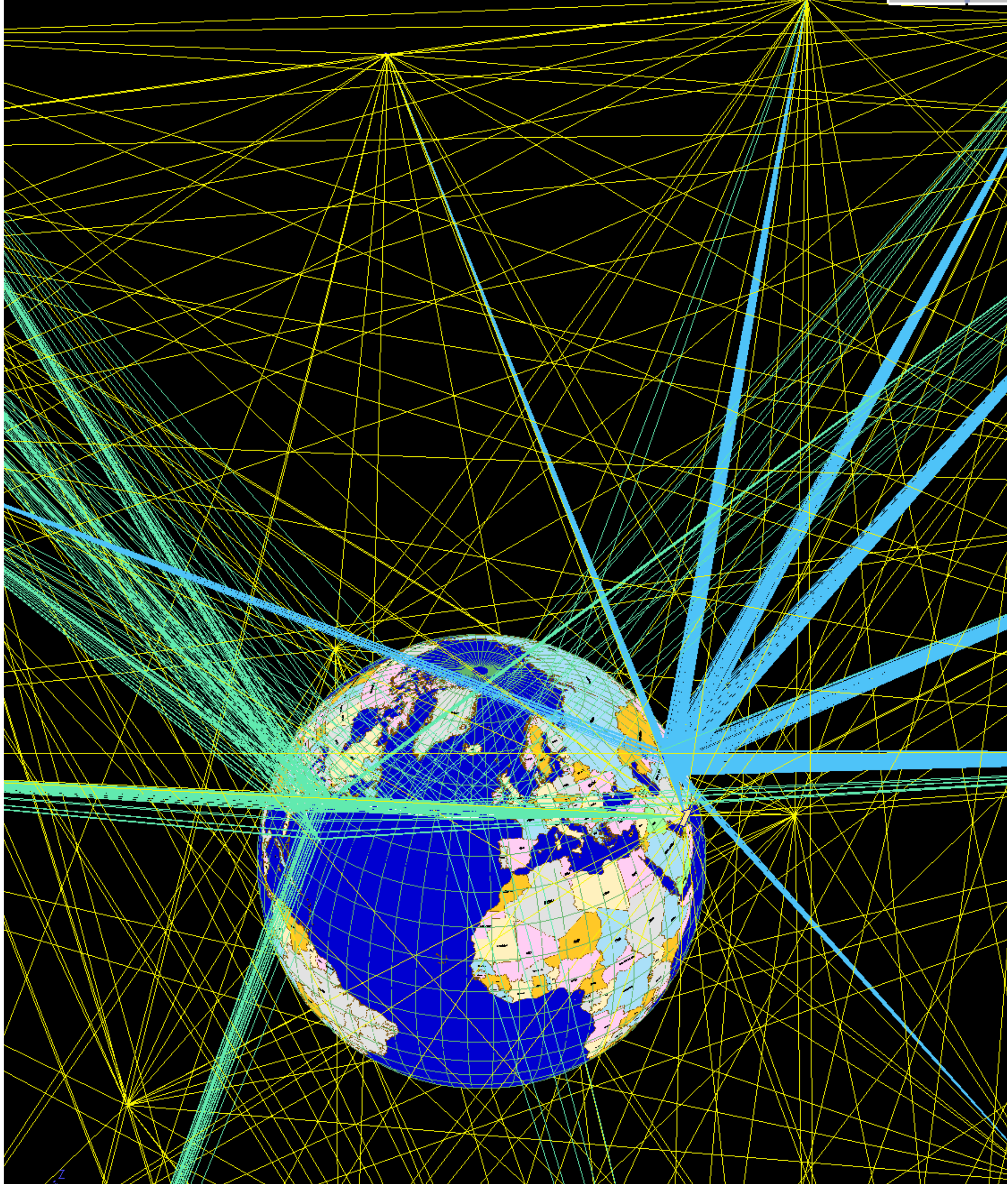

Figure 18. Connectivity between satellites, ships, aircraft, and ground vehicles.

In addition, each box has an Altitude coordinate that is perpendicular to the surface of the earth and as high as necessary for the application. All points above the earth's surface are described in terms of (LAT, LON, ALT). This hardware architecture uses 27 Green Gene (GG) boxes, see [23], with the box in the center performing management control functions.

Figure 19 provides additional boxes around those modeling the earth's surface and the space above it to model spatial platforms (e.g., satellites) that are connected to greater areas. Adding 5 boxes around the periphery is generally sufficient, yielding a total of 32 GG boxes on one tightly coupled GG rack.

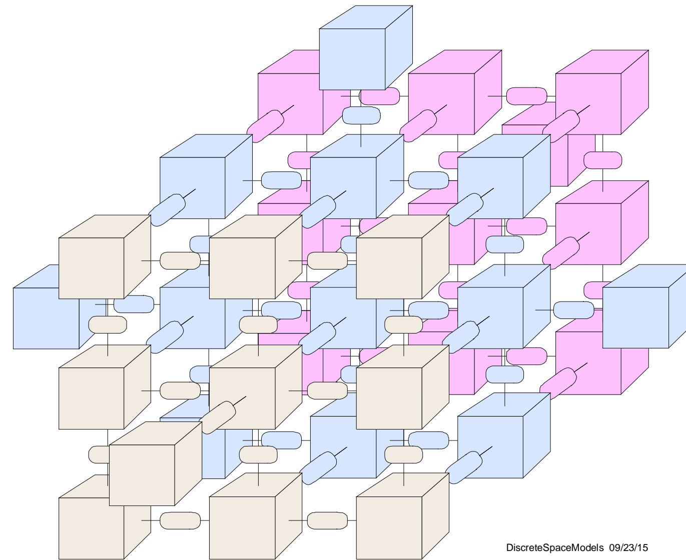

Figure 19. Representing the earth's surface and the space above it.

SUMMARY

This paper addresses the major problem with parallel processing performance as described by top engineers who understand the need for a new approach to software. It describes a CAD system that is proven to maximize module understandability and independence, eliminating typical exponential growth of complexity, and supporting direct use by application experts. It provides the Application Space Architecture (ASA) to greatly simplify software design for parallel processors, producing run times that are orders of magnitude faster on single as well as parallel processors. The ASA is the ISA equivalent for parallel processor designs.

This architectural approach applies to a broad spectrum of applications running on simplified hardware architectures. To accomplish this, the languages support large data spaces with deep hierarchies to simplify complex transformations that provide huge increases in speed. Application area experts work with engineering drawings to visualize the independence of modules and optimize the architectures to run concurrently on large parallel processors.

This CAD system supports rapid understanding of the underlying principles necessary to substantially simplify software development on parallel processors. Without hardware designed based on the ASA, it is difficult to take total advantage of the inherent parallelism in applications that are not embarrassingly parallel.

Having used and tested the ASA, it becomes apparent how the number of processors required to meet application speed constraints can be reduced by 2 to 4 orders of magnitude, achieving huge economic benefits for its users. This system has evolved under substantial experimentation on many projects over many years. Experiments are easily repeated by others.

References

[1]. Bailey, D.H., Twelve Ways To Fool The Masses When Giving Performance Results On Parallel Computers, Supercomputing Review, Aug. 1991.

[2]. Beyer, Kurt W., *Grace Hopper and the Invention of the Information Age*. Cambridge, MA: The MIT Press, (2009). ISBN 978-0-262-01310-9.

[3]. 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.

[4]. Cave, W.C., *High Efficiency, Scalable, Parallel Processing*, DARPA Contract SF022-035 Final Report, Prediction Systems, Inc., Spring Lake, NJ, June 2003.

[5]. Cave, W.C., et.al, **Software Theory For Parallel Processors**, Visual Software International, Sept 2015, Spring Lake, NJ, www.VisiSoft.com/SoftwareTheoryBook.pdf .

[6]. Cole, Bernard, EE Times Article, Sep 13, 2007, www.eetimes.com/showArticle .

[7]. Cusumano, Michael A., "What Road Ahead for Microsoft and Windows?," Communications of the ACM, July 2006, pg 21-23.

[8]. Dyson, George, **Turing's Cathedral**, Pantheon Books, New York, NY, 2012.

[9]. *General Simulation System - User's Reference Manual,* Version 10.6, Visual Software International, Spring Lake, NJ, 1984-2015, www.VisiSoft.com/GSS_Users_Manual.pdf .

[10]. Kambayashi, Yasushi and Henry F. Ledgard, "The Separation Principle - A Programming Paradigm" IEEE Software, March/April 2004

[11]. Krishnadas, L.C., EE Times Article, Jan 17, 2008, www.eetimes.com/showArticle .

[12]. Ledgard, H., et al, "The Natural Language of Interactive Systems," CACM No. 10, October 1980, pp 556-563.

[13]. Leopold, George, "IEEE Group Seeks to Reinvent Computing as Scaling Stalls," HPC Wire, May 7, 2015, www.WeeklyUpdate@hpcwire.com .

[14]. Merritt, R., Multicore puts screws to parallel programming modules, EE Times On Line, Feb 15, 2008, www.eetimes.com/showArticle.jhtml?articleID=206504466 .

[15]. Mills, H.D., Mathematical Foundations of Structured Programming, Technical Report FSC 72-6012, IBM Federal Systems Division, 1972.

[16]. Ritchie, D.M., AT&T Bell Laboratories Technical Journal, pg 1591, Vol. 63, No.8, Oct 1984.

[17]. Schweppe, F., **Uncertain Dynamic Systems***,* Prentice Hall, Englewood Cliffs, NJ, 1978.

[18]. Shannon, C.E., a mathematical theory of communication, BSTJ, Vol.27, pp 379 & 623, Jul & Oct 1948.

[19]. 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 .

[20]. Trader, Tiffany, "Horst Simon on the HPC Slowdown," HPC Wire, February 13, 2015, www.WeeklyUpdate@hpcwire.com .

[21]. van der Linden, P, **Expert C Programming - Deep C Secrets***,* SunSoft Press - Prentice Hall, Englewood Cliffs, NJ, 1994.

[22]. Wakabayashi, Daisuke, "Microsoft's Top Visionary Sees Parallel World", Reuters, Seattle, March 13, 2008. http://in.reuters.com/article/2008/03/13/microsoft-mundie-idINN1222563020080313

[23]. Website for the Green Gene Machine: www.GreenGeneMachine.com