OVERVIEW

The need for an engineering approach to overcome barriers when dealing with increased software complexity is described in various papers, see [1], [2], [3], [10], [11], [14], [24] and [25]. This need is particularly obvious when dealing with applications requiring high speed multipliers using parallel processors, see [17], [19], and [28]. Two papers, [6] and [21], call for Computer-Aided Design (CAD) tools for software, and a corresponding engineering approach similar to that used in hardware design. This paper presents a CAD approach to building software for a wide variety of applications, including a parallel processor OS that substantially simplifies developing software with high Processor Utilization Efficiencies (PUEs).

To fairly compare software development environments, one must consider the following factors relative to the target applications.

- Level of software complexity
- Time and cost to develop and support the application
- Time and cost to learn and use the application
- Use of parallel processors to meet run-time speed constraints

As complexity grows, a barrier to enhancement is created as a heavier burden is placed on the development environment. This burden directly affects control over the design as well as developer productivity, especially when using parallel processors. Understandability is a critical aspect of complex systems where detailed knowledge lies with subject area experts. To linearize the growth of complexity (versus growing nonlinearly) requires a system to be decomposed into modules that are maximally independent. To achieve high levels of understandability and independence requires a new approach to building software. This is accomplished by VisiSoft, a CAD system for developing software that is directly understood by subject area experts.

APPLYING ENGINEERING PRINCIPLES TO SOFTWARE

The CAD approach to software described here follows from Shannon’s Mathematical Theory Of Communications, [23], also known as Information Theory. Based on the binary number system, this theory defines 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. This CAD approach also follows from the State Space framework formulated by control theory engineers to simplify complex problems in control system design. The key to simplification is selection of the best space to support the transformations. State Space extends the mathematics of vectors and matrices, simplifying complex transformations using large vector spaces.

† Visual Software International - www.VisiSoft.com ; *University of Toledo
**Software Spaces & Databases**

Simplification of complex mathematical problems hinges on selection of a good space, reducing complexity of the transformations and the corresponding time to solve the problem. This is apparent when dealing with multi-dimensional spaces as occurs with software. Software spaces are determined by the databases used to support instructions (transformations) that implement the dynamics of an application. One can view the entire database as the overall software space, containing independent subspaces that support specific actions or transformations implemented by sets of instructions.

Figure 1 illustrates the development of software systems. 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 problem into a software space that best suits their application. Best implies simplicity of transformations and minimized processor counts to meet 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.

**Software - An Extension Of Mathematics**

As shown by numerous experiments, [9], simplifying software design is accomplished using the same basic approach for solving complex mathematical problems, i.e. to find the space that provides the greatest simplification of the equations. The critical property underlying this approach is selection of the best set of independent coordinates.

If a software application is represented by a continuous-time or discrete-time linear mathematical model, the software and mathematical solutions are essentially 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 [15]. 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.,

```plaintext
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. More importantly, they will likely contain statements that SCHEDULE a NEW_EVENT in the future. This may be used in a time-based model or a real-time system.
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. The major objectives of communication systems are reliability and speed. Fast and reliable transfer of information is the goal of information theory, as evolved by Shannon, [23]. The basic principle is that: Reliability of information transfers is increased by adding redundancy (i.e., additional data). This is used to write and read computer memory where code bits are added to the data being stored, decreasing the probability of error when reading it. In wireless 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 information being transferred reliably and of 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,” see [26]. It dominates the world of free trade. Considering the small islands where it originated, its survival is attributed to the success of its users - thanks to its reliability.

Studies comparing interactive languages have shown that errors increase as statements move from good English to a more terse form, see [18]. Comparisons of COBOL, FORTRAN and C-based languages will typically derive the following conclusions: COBOL is verbose; FORTRAN is fair; C-based languages are terse (an objective of the principle designer).

Generalized State Space

Generalized State Space capitalizes upon the concepts of the State Space framework used in control theory by extending the mathematical definitions of vectors and transformations, see [13] and [22]. 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 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 transformations (instructions). This framework for designing software is called Generalized State Space, see [7].

Hierarchical Data Spaces

Complex data structures are more easily understood when put into a hierarchy, see Figures 2 and 3. Deep hierarchies allow large complex data structures (resources) to be moved in a single instruction fetch, with all of the individual fields directly available to instructions (processes) that use them. This supports order of magnitude improvements in single and parallel processor speeds. Figure 3 illustrates an “Instanced” resource supporting multiple transmitters and receivers. Not shown are the QUANTITY clauses used in Figure 2 to define tabular arrays. Multiple TRANSMITTERs and RECEIVERs are identified automatically when invoking the instructions that use the resource. This is because they are part of an instanced module.
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 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

RESOURCE: INDEXED_MESSAGE_TABLE

RESOURCE NAME: 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_LOSS REAL
    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 GOODCEPTION CONFlicTINGCEPTION CONFlicTING.Broadcast

Figure 2. Example of a hierarchically structured Resource.

Figure 3. Building hierarchical data structures using the Resource Language.
In the resource in Figure 3, 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 subject area experts.

Hierarchical Transformations

Figure 4 is a VisiSoft Process. The hierarchical organization applies directly to simplification of complex instruction sets using one-in one-out control structures, see [20]. With automatic instancing, subscripts corresponding to quantity clauses in Figure 2 are unnecessary.

```
PROCESS: RECEPTION
INSTANCES: TRANSMITTER
RESOURCES: TRANSCEIVER
RECEIVER
INSTANCES: TRANSMITTER
MESSAGE_FORMATS
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 4. Building hierarchical rule structures using the Process Language.

As described by Shannon, [23], the language used to transfer information plays a major role in ensuring its correct reception. As shown by Ledgard, [18], 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 [5].
When building complex software, human translation is simplified if a language supports obvious representation of physical behavior. The examples in Figures 2, 3 and 4 are taken directly from large simulations of complex 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 4. Actual systems may entail more complex resources and processes than those shown, but are easily understood by subject area experts.

The languages of this CAD system are designed to simplify mapping application requirements into a software space. The resulting computer translation becomes much more complex, exactly where the burden should be, see [27], pg 64. Engineers concerned with modeling and simulation of complex systems can easily represent their problem in the 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, see Chapter 17 in [9].

The Separation Principle

The underlying principle supporting the visualization of software design 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), [15], this has become known as the Separation Principle, [16]. 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 5. Processes containing instructions that implement transformations are depicted as rectangles. The lines connecting them determine which processes have access to what resources. Each process has a dedicated resource and two 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.

![A SEQUENCE OF TRANSFORMATIONS](image)

Figure 5. State vectors and transformations.
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 *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 6. 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. By grouping icons into hierarchies of modules, module independence can be visualized directly.

**SOFTWARE ARCHITECTURE**

**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 6. Only then can one visually observe independence - the key property supporting concurrency on parallel processors. Flow charts, or graphical variations on flow charts, are of little use when describing the property of independence.

Hardware engineers are concerned with reliability and maintainability of modules that can wear out or be damaged. This requires the ability to replace a module without affecting the rest of the system. To do this, modules must be designed to be independent, allowing them to be replaced without change to other modules. When modules are independent, system complexity increases linearly (instead of exponentially) with growth. This increases productivity in development as well as when enhancing the system.

Decomposition of complex systems into independent parts requires the knowledge of a subject area expert. Figure 6 illustrates a decomposition following the organizational lines of the application (predicting electro-magnetic wave propagation in terrain). Both the architectural decomposition and design of the individual modules require subject area expertise.
Figure 6. Illustration of editing resources and processes on the engineering drawing.
In VisiSoft, which processes have access to what resources is determined solely by the architecture - not the code. This includes the manner in which data is shared (there is no global data in VisiSoft). The languages do not permit declaration of scope rules, and all data is automatically accessed by pointer. 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, understandability of complex software is difficult. Their value is most apparent when building software for parallel processors, where modules must be independent to run concurrently on separate processors. These properties provide a huge improvement in the approach to developing software.

Visualization

Most engineering fields (e.g., Architectural, Aeronautical, Electrical, Mechanical, etc.) would be at a huge disadvantage without engineering drawings providing precise descriptions of connectivity. Software is no different. But one must have first hand experience using them to understand their importance. Engineering designs also require written (language) specifications. Typically, the architecture versus language crossover point is obvious, as it is in software. But without languages that separate data structures from rules, and support deep hierarchies of both, independence and understandability are lost. Without these properties, visualization of architecture, modularity, and the advantages of engineering drawings cannot be realized.

Without drawings, sharing of data is often driven by the desire to save memory. When sharing variables, dependent interconnections are common, and modification of one part of a system may affect other parts that were thought to be independent, see [12]. With the design in Figure 6, independent modules may be refined or replaced without changes to the rest of the system. Most importantly, memory need not be shared or moved, yielding substantial increases in speed, especially on parallel processors. This approach uses more memory - the most abundant resource - yielding solutions that provide order-of-magnitude reductions in run time on a single processor, and substantially simplifying the use of parallel processors. This results in Processor Utilization Efficiencies (PUEs) between 75% and 95%, see Chapters 17 and 18, [9].

Understandability & Independence

As indicated above, understandability must be constantly improved so that people with less experience, or those concerned with only part of the system can be productive. This implies that a system can be decomposed into parts that are relatively independent. When modules are independent, they are also much easier to understand. Figure 7 illustrates that understandability is not orthogonal to independence. Understandability increases with independence. These two interrelated properties are key to dealing with complexity.

Figure 7. The critical factors affecting the ability to deal with complexity.
Implementing understandability and independence depends directly on the language used to produce the code. It must be highly readable by subject area experts and support substantial layers of hierarchy of both data and instructions. The language must also support architectural facilities that provide direct recognition of the connectivity of modules through visualization. These simplifications can only be achieved when the language is tailored to the architectural properties necessary to create an easily understood hierarchical organization.

Hierarchical Modules

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, both in architecture and language. They support the specification of complex data spaces that are used to simplify complex algorithms. The number of levels in a hierarchy must be sufficient to push down the complexity, making the organization of the system easy to understand.

As systems become more complex, it is necessary to create hierarchies of modules, where the number of levels in the hierarchy will be determined by the size and complexity of the system. The higher the complexity, the deeper the layers of hierarchy required. The manner in which a system is decomposed into modules will depend directly on the application. Design of the overall hierarchy as well as the sub-hierarchies will determine the understandability of the overall system and the time to develop and enhance a reliable product. The resulting design will depend heavily upon the nature and functionality of the application system itself, and upon critical contributions from subject area experts.

As illustrated in Figure 6, 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 6 is a Library module that is sufficiently complex to warrant its own drawing. As illustrated, this CAD system allows resources and processes to be edited directly on the drawing.

PARALLELISM, ARCHITECTURE, AND DECOMPOSITION

When striving to take advantage of the inherent parallelism in a system, one must determine the architecture of software modules that maximizes concurrency on a parallel processor. Decomposing the overall space into an appropriate set of resources is key to achieve simplified transformations, and minimized processor counts to meet run-time constraints.

Discrete Event Models

The VisiSoft CAD system was developed initially to simplify discrete event simulations running on parallel processors to support design of communication systems. This presented two requirements. First is the need for complete and consistent results when using parallel processors. This implies that two processes must be independent in order to run concurrently on two different processors. To be spatially independent, these processes must not share any data. To be temporally independent, they must not share data directly while they are both running (further defined below).
Using a discrete event model, the need to schedule a future event is determined when a particular action is taken. Scheduling a future event 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 some flag has been set. Each of these actions wastes considerable time. Using discrete event models, the event is performed when popped at its scheduled time, a facility provided by the VisiSoft Parallel OS (VPOS). One can also cancel events prior to their scheduled time.

Next is the need to minimize memory management overhead due to swapping processes and paging data. This is accomplished by maximizing the work done by each process while running concurrently with processes on other processors. VisiSoft Independent (IND) modules are used to represent modules that can run concurrently, where tiling of subscripted computations is a small subset.

Software Languages To Support Parallelism

Requirements for the resource and process languages were driven in part by factors similar to those motivating the use of tiling in parallel versions of FORTRAN. These factors minimize memory management overhead due to swapping and paging. This is accomplished by maximizing the work done on each processor - while running concurrently with work on the other processors, thus maximizing the PUE.

To do this, the language 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, likely the most knowledgeable software language designer, [5], such organizations are best supported by deep hierarchies of both data and instructions.

Inter-Module Communications

Independent (IND) Modules residing on separate processors must communicate with each other. Design of the resources used to communicate between IND modules must ensure their independence and ease of understanding of the module designs.

Communication between IND modules residing on different processors is most easily accomplished using a simplex channel. Two-way communications use a pair of simplex channels. Experience with architectural design for parallel processors shows that Inter-Processor (IP) resources are best written by only one module, and only read by IND modules on other processors. This is because the 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, they 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 VisiSoft, synchronization of IP resources is done automatically using copies of memory.
Parallel Processor Architecture Requirements

To simplify software development on parallel processors, 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 modules that are independent, i.e., they only share data using IP resources. To determine the independence of modules, designers must be able to easily see which processes share what data (IP resources). This can only be done when the following critical requirements are met:

- Resources can be organized into the deep hierarchies required to represent the best spaces to implement problem solutions;
- 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 to assure module independence.

The above requirements are met when the data language supports large data structures using deep hierarchies. They also require an instruction language that supports large hierarchies of rules. Both looping and complex IF ... THEN ... ELSE statements are then flattened. What is known as *Waterfall* or *Fall through* code is gone (with no GOTOs). These properties dramatically simplify design of the best data spaces, and concurrently, the design of complex algorithms. Both lead to substantial increases in both understanding and run time speed - on single as well as parallel processors, see [8] and [9].

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 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 8: one for resources (data structures); one for processes (rule structures); 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 automatic allocation of parallel processors. When the Control Specification is translated, Run-Time System (RTS) code 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. All can be edited from the drawing. Figure 8 illustrates a large simulation used for military planning, containing many types of fast moving platforms within multiple IND modules. Numerous experiments have been conducted using this system on a parallel processor. These are described in [9].
Figure 8. Example of an engineering drawing of parallel processor software.
Referring to Figure 9, if sufficient inherent parallelism exists in the system, architects can decompose software into large IND modules. As described in [9], 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.

**Figure 9.** The parallel processor software-hardware environment.

### Temporal Independence And Synchronization

The critical property determining complete and consistent results of the transformations, and the ability to run concurrently, is that of independence. This implies that processes are spatially independent when they share no resources. This property - two processes not connected to the same resource - represents the spatial independence of the processes. When processes are connected to the same resource but inhibited from running concurrently, e.g., on a single processor - or through synchronization on a parallel processor, they have temporal independence. Synchronization is automatically invoked by VPOS using information provided by the Run-Time System (RTS).

As stated above, IND modules communicate using IP resources. IP resources may be read or written by any process within the same IND module, but processes outside that module may only read them. 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.

All IND modules are automatically synchronized in time within a user-specified $\Delta T$ Time Interval by the VPOS to ensure that the results are complete and consistent, see [9]. $\Delta T$ is defined by the designer in the Control Specification based upon the 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 by VPOS. This allows two IND modules to run concurrently.
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. A Run-Time System (RTS) is generated from that information to control VPOS calls that assign modules to processors. 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 [9].

FAIR COMPARISONS BASED UPON EXPERIMENT AND MEASUREMENT

When conducting experiments to compare software languages and development environments, one must avoid the potential pitfalls that produce biased results, see Bailey, [4]. There are multiple causes. 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 10 shows the results of experiments manipulating a large database where different spaces (data structures) were used to implement the design, see Chapter 17 in [9]. The differences in speed using 5 different approaches show a factor of 100. If one comparison uses test 1, the parallel processor speed multiplier will be 100 times faster than one using test 5. This points out the concerns of Bailey, [4]. Alternatively, using the approach in test 5, one may cut the number of processors by 100 to achieve the same speed multiplier.

Figure 10. Single processor speed multipliers using different design spaces.
Furthermore, using the VisiSoft environment, one is likely to obtain a PUE between 75% and 95%, compared to the 7% to 10% typically achieved by current approaches, see Chapter 18 in [9]. This provides another factor of 10, a direct multiplier on speed. This coupled with the single processor speed gains of 10 to 100, one can gain an overall factor of 100 to 1000. Now consider that a 32 processor PC can be used to meet a speed constraint requiring a 3000 processor HPC. Just the reduction of distance between processors could gain another factor of 10 increase in speed. So combining these factors, one can conclude that a 32 processor PC could beat a 3000 processor HPC when properly compared in a fair test.

SUMMARY

This paper addresses the major problem with parallel processing as described by the top engineers at Intel, AMD, and Microsoft - the need for a new and proven engineering approach to software. It describes a CAD system that maximizes understandability and independence, eliminating typical exponential growth of complexity, and supporting direct use by subject area experts. It provides facilities that greatly simplify the design of software for parallel processors while producing run-times that are orders of magnitude faster on single as well as parallel processors. To accomplish this, the languages support large data spaces with deep hierarchies to simplify complex transformations that provide huge increases in speed. Users work with engineering drawings to visualize the independence of modules and optimize the architecture so they can run concurrently on large parallel processors. This system has evolved under substantial experimentation over many years, and the experiments are easily repeated by others. This CAD system supports rapid understanding of the underlying principles necessary to simplify software development on parallel processors. It reduces the number of processors required to meet application speed constraints by orders of magnitude, achieving huge economic benefits for its users.

REFERENCES


