FINAL REPORT

High Efficiency, Scalable, Parallel Processing

for

DARPA (IPTO)

SBIR CONTRACT: SB022-035

June 30, 2003

PREDICTION SYSTEMS, INC.
PREDICTION & CONTROL SYSTEMS ENGINEERS

309 Morris Avenue
Spring Lake, NJ 07762

(732)449-6800 (732)449-0897
psi@predictsys.com www.predictsys.com

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TITLE	3. DATES COVERED (From - To)
06/30/2003	FINAL	10/8/2002 - 06/30/2003

4. TITLE AND SUBTITLE

High Efficiency, Scalable, Parallel Processing Approaches for Multi-Sensor Data Fusion

5a. CONTRACT NUMBER

DAAH01-03-C-R031

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Robert Wassmer

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Prediction Systems, Inc.
309 Morris Ave.
Suite G
Spring Lake, NJ 07762

8. PERFORMING ORGANIZATION REPORT NUMBER

PSI03004

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

US ARMY Aviation & Missile Command
AMSAM-AC-RD-AX
Beverly Gonzales
Redstone Arsenal, AL 35898-5280

10. SPONSOR / MONITOR'S ACRONYM(S)

11. SPONSORING / MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT

Unlimited

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

PSI's CAD approach to simulation / software development cuts large system life cycle costs by an order of magnitude. A visual representation of software architecture that provides a one-to-one mapping to the code, it's based upon separation data from instructions, affording separation of architecture from language, and providing ease of control and reuse of complex modules. This paradigm shift for software brakes barriers to building complex systems, makes significant upgrades easy, and cuts support costs dramatically, by achieving module independence through visually enforced architectural design rules. Its success has resulted in huge simuatlions that meet customer validity contraints but now exceed single processor computer power by one or two orders of magnitude. Hardware designers produce parallel computers with speeds into teraflop ranges. However, their practical use on all but very special problems has been extremely limited due to software implementation problems. This research is to confirm hypotheses that graphical design rules that achieve software understandability and module independence also support allocation of processes to parallel processors. Fusion of this technology with parallel processors can result in an order of magnitude of speed improvement, yet making development easier on parallel machines than single processor machines due to concurrent memory access and management.

15. SUBJECT TERMS

Computer-Aided Design (CAD)
Software
Simulation
Parallel Processing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSBILE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			William C. Cave
unclassified	unclassified	unclassified	unclassified	80	19b. TELEPHONE NUMBER *(Include area code)* 732/449-6800

TABLE OF CONTENTS

1. BACKGROUND

Hardware designers have succeeded in producing parallel and distributed processor computers with theoretical speeds well into the gigaflop range. However, the practical use of these machines on all but some very special problems is extremely limited. The inability to use this power is due to great difficulties encountered when trying to translate real world problems into software that makes effective use of highly parallel machines. This has been described by numerous authors over many years, see for example [1], [2], and [3].

COMMERCIAL MARKET REQUIREMENTS

In the commercial marketplace, speed benefits gained using a parallel computer must sufficiently outweigh the cost to develop and support the software. If not, then real commercialization, based upon solid economics, will not occur. These economic goals will be achieved if the following requirements can be met:

1. Subject area experts who understand the problems to be solved must be able to describe them easily and directly to computers without concern for parallelism, or even prior knowledge of computer programming.
2. The software must be generated automatically to take full effective advantage of the inherent parallelism of the problem on a Massively Parallel Processor (MPP).

These two requirements are tightly interrelated. The subject area expert should not care whether the problem is being solved on a single processor machine, or one with hundreds of processors. The run-time software must be generated to make effective use of the available parallelism of the host machine, adapting to changes in the environment, a very tedious but mechanical process.

REQUIREMENT FOR SPECIAL PROGRAMMING SKILLS

Current approaches to solving problems on parallel processor machines have not, in general, overcome these two barriers. Problem description for parallel - as opposed to single - processing generally incurs a huge cost increase for all but a few special cases. This is compounded by the fact that the problems requiring large processor power are themselves complex, and best understood by subject area experts.

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

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

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

MULTI-SENSOR FUSION APPLICATION

PSI currently has a contract with the Army CECOM to develop an Operations Management System (OMS) for the Netted Full Spectrum Sensor (NFSS) system, [4]. This system must contain an embedded simulation of the detailed target environment, electromagnetic environment, sensors, communications, as well as actual multi-level/multi-sensor fusion algorithms and control systems to predict the responses to multiple sensor tasking. It also contains an optimization subsystem to determine the most effective use of sensor resources during large operations. In addition to other challenges, this represents a multi-faceted high stress computational problem. This system has been used as the basis for analyzing and evaluating approaches to parallel processing under the Phase I contract.

2. INTRODUCTION

Solutions to the parallel processing problem tend to skip over the software piece of the problem, going from application requirements to hardware architecture. (The word *architecture* implies hardware in the parallel processing literature. The words "software architecture" do not appear.) Software is not much more than an afterthought relative to the size of the hardware design effort. This approach, illustrated in Figure 1, is termed *software bypass*.

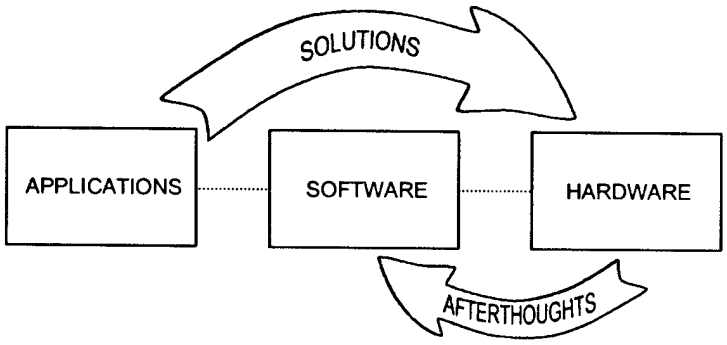


Figure 1. Software bypass - designing the hardware first.

Subject area experts who want to use parallel computers cannot simply enter their problem specifications into a piece of hardware. They must first write the very complex software required to control parallel processor hardware. Without knowledge of the special operating systems and languages for parallel computers, these experts typically turn to programmers to do the job. Programmers see the chance to increase their value by learning how to be parallel programmers. Their interest is in learning deeper specializations to broaden their higher-paying job opportunities. This cycle of thinking is at odds with commercial market requirements.

USE OF ABSTRACT REPRESENTATIONS

Certainly there are many uses of abstractions when building models of highly complex systems and their environments. One could not perform simulation without abstraction of reality into models that run on a computer. The General Simulation System (GSS), [5], provides for ease of abstraction where complex processes that may be spread across all of the entities in a system are represented in a single list. GSS contains a library of high speed list management facilities that eliminate the need for the modeler to develop linked list software, a basic abstraction in modeling. However, one must consider the trade offs between time and cost of development as well as speed and memory utilization at run-time.

With today's parallel processors, memory utilization is not an issue. It is difficult to conceive of a problem where the amount of memory on a large parallel processor computer presents a limitation. Using conventional techniques for parallel processing, the trade is usually between development time and running time, given resource constraints in dollars. This leads to decisions on how models are represented. The choice is usually between the way one deals with abstractions, and typically ends up with substantial hand tailoring of code to the parallel processing environment. This implies a huge effort in development, resulting in significant time and cost, to use parallel processors. More importantly, the abstractions required for parallel processing make it difficult for a modeler with subject area expertise to understand the code.

THE INHERENT NATURE OF SYSTEM DECOMPOSITION

As systems are designed to be more user-friendly and adapt to their environment with greater effectiveness, they become more complex. To deal with a high level of complexity, designers must partition systems into modules that operate independently, minimizing the shared interfaces. If module interfaces are designed for maximum isolation, they incur a minimum transfer of information. This maximizes the ratio of internal processing to interface processing, which in turn maximizes their measure of independence. This is the type of software architecture required for effective use of parallel processing. Given a high degree of module independence and inherent parallelism, many applications have still failed to achieve a high degree of efficiency in parallel processor utilization. This is because current software approaches cloud this level of architecture.

The two most prominent parallel processing companies in the early 1990s, Kendall Square Research (KSR) and Thinking Machines Corp. (TMC), failed due to lack of good software environments for both developing and running applications. There are a number of reasons that no software environment has yet to crack the problem. We believe that the two most important reasons are:

(1) Decomposition of a large software system is an architectural problem, and the architecture of a system of independent modules is best described graphically (like hardware) - not using a language;

(2) Software architectural design methodology and supporting technology have not been tied to the requirements of efficient scheduling and assignment of processors to processes during run time.

After one gains a good understanding of the software side of the parallel processing problem, it becomes clear that the *language environment* must be designed to support the *architecture environment* as well as the requirements for understandability and independence of the detailed implementation. This has major implications on *scoping the size* and *controlling the hierarchies* of independent modules. At least as important, the architecture environment must serve to optimize the scheduling and assignment of processors to processes in the *run-time environment.* Our proposed solution solves both problems.

3. PERTINENT CONSIDERATIONS

Future survival depends upon the speed with which one can deal with increasing complexity.

THE IMPACT OF SPEED AND COMPLEXITY ON SURVIVAL

The things we take for granted today would have boggled the minds of people just 100 years ago. Looking back 1000 or 10,000 years is awesome. Which way would any of us prefer to live? Who is better prepared to survive? The answer to the first question is generally obvious. The answer to the second requires more consideration.

The U.S. is learning that there are many faces of survival. The days of firearm versus bow and arrow are long past. Yet a high speed aircraft with smart missiles may not help preserve our own infrastructure when attacked by terrorists. The approach to survival is taking on a different meaning than historic war. The enemy situation is becoming much more complex. Accurately predicting what an adversary may do depends upon how much time he has to think, communicate, and take action. The problem of defending the U.S. is being redefined in light of the increasing need to deal with speed and complexity as we endeavor to survive.

Dealing With Increasing Complexity

Anyone familiar with the history of mathematics knows the motivations leading to the progression of numbers. It started with "whole numbers" or *integers*, and progressed to *signed integers,* then to *fractions* and *rational numbers*. It continued to *real numbers*, *imaginary* and *complex numbers*. Each step covered a more complex realm - not by imagination, but by necessity.

There is more to this progression than just the increase in complexity. Each of these extensions is still referred to as a number. And each encompasses the prior. Real numbers are a subset of complex numbers. More importantly, many of the laws and transformations still apply as we move up the scale of complexity. Their interpretations are simply extended to be more general. This allows us to deal with jumps in complexity.

Selecting The Most Convenient Coordinate System

As we continue to move up the food chain of numbers and mathematics, we can group numbers into *vectors*. The position of a body in space can be described by three numbers depending upon the coordinate system we choose. And we learn in higher levels of mathematics and physics, particularly in electro-magnetic theory and partial differential equations, that problems can be solved more easily if we select the right coordinate system. For example, when a particle moves in a spherical orbit, it is much easier to describe its motion in spherical coordinates. Cartesian coordinates will work, but it takes longer to solve the problem.

Selection of the most convenient coordinate system is typically taught under the topic of *separation of variables*. One learns that the separation principle can be used if the variables form an *independent* set. The property of independence can be verified using specified tests. The concept of choosing the best coordinate system and the property of independence are the important principles one can apply when dealing with complexity in a constrained time environment. We will make use of these concepts.

Einstein introduced the use of *tensors* to deal with the increasing dimensions of time, velocity, and acceleration. Control system engineers developed the *state vector* to account for the many degrees of freedom required to characterize complex dynamic systems. The *state space* framework has been shown to be the most general representation of a dynamic system, see [6], and [7]. Providing a framework for problem description was not the only benefit of the state space approach. It also afforded a framework for developing faster solutions to problems that could run for days on the computers of the time.

FRAMEWORKS FOR REPRESENTING COMPLEX DYNAMIC SYSTEMS

In a competitive time-constrained environment, time (speed) is the most important factor. If two sides develop the same capability, the one that gets there first is likely to be the one that wins. When building tools to help people solve design problems or make complex planning decisions, time enters into the picture in at least two major ways.

- Development Time - the time it takes to develop the tool
- Solution Time - the time it takes to get a useful solution from the tool

One can imagine a great tool for solving a problem. But one must answer the question - can we get it built in time to accomplish our goal? Or, more importantly, will it produce valid answers *fast enough* if we get it built? Of course cost and risk are also major factors. However, time is usually of the essence.

Automating The Representation Process

In the early 1960's, electronic circuit designers developed automated tools for solving complex systems of nonlinear differential equations required to represent digital waveforms in the time domain. These Computer-Aided Design (CAD) tools allowed engineers to describe large networks topologically and write FORTRAN-like equations describing nonlinear functions. Programming skills became unnecessary. The code needed to generate and run simulations of very large networks was generated automatically. This afforded a huge leap in design productivity. It enabled the design of huge complex networks leading to integrated circuit design.

CAD system development became a business for many, including the principals of PSI. Two systems were developed, one for continuous system modeling (e.g., for digital circuit design), and one using a discrete-time framework (for the design of signal processing systems). The second used sampled data principles to reduce computation time. An underlying *state space* framework supported both products.

For large networks, the number of state variables runs to thousands. Solving worst case design problems involves multiple optimization runs of thousands of simulations. Each simulation has to solve the optimal control problem, involving thousands of nonlinear differential equations. Speed and accuracy are the driving forces in designing these systems. If it takes a computer days to get a design, only one or two test points are produced in a week - not very attractive.

Capitalizing Upon General Principals

State space is used because it provides the most convenient framework for solving any type of dynamic problem. The general form of the solution holds for any set of independent state variables. This allows for the development of generalized methods, e.g., optimal sparse matrix inversion and describing functions, to solve nonlinear problems fast while ensuring algorithm convergence. The end result is to solve huge problems in minutes. However, this approach requires formulating problems in a mathematical framework.

Facing Totally New Problems

In PSI products prior to 1982, models were formulated mathematically, i.e., using vectors, matrices, and systems of equations. This approach allowed the solution to be derived automatically and solved very fast. By 1982, this approach was recognized to have severe limitations when modeling communications or control systems involving algorithmic decision processes. Clients wanted to describe their problem using more general state concepts, and be able to write conditional statements within the system of equations. It was determined that these types of decision processes could be handled using the discrete event approach originally developed by Gordon in 1961, see [8] and [9].

A MORE GENERALIZED PROBLEM FORMULATION

In 1982, discrete event simulation was analyzed. The motivation was high because of the requirement for writing decision algorithms into the models. Users wanted to break up systems of equations and embed English-like conditions and rules, e.g.,

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

Additionally, there were complaints about the inability of existing discrete event simulation products, e.g., GPSS, SIMSCRIPT, and SLAM, to solve our client's problems. The major complaints were lack of scalability (inability to deal with increasing complexity) and excessive simulation run-times. This led to an investigation of the deficiencies of the other products in the market, as well as an analysis of how to formulate the basis for general solution.

At first it appeared difficult to derive a mathematical framework to support this new requirement. This caused concern about the ability to justify design decisions without a formal framework. We appeared to be leaving the world of mathematics. Time steps were determined by the modeler in terms of scheduled events. This led to the development of a state space definition of discrete event systems. A description of this is provided in the Sections below entitled Concept Of A Generalized State Vector and State Space Definition Of A GSS Model. The differences and likenesses of mathematical and rule oriented formulations are compared in *Simulation Of Complex Systems*, [10].

Facing The Speed Issue

Because of the excessive running times of competing products (some critical simulations were taking 5 to 7 days to run a 2 hour scenario), it was determined that if PSI developed a new product, it must be able to run on a parallel machine. PSI's experience in computer design, parallel processing, and the knowledge of how chips were evolving to support fast computing methods led to an approach that would take advantage of future hardware technology.

Parallel processing imposes the requirement that two or more processes must run concurrently on separate processors. This implies that concurrent processes must be independent. The property of independence implies that the processes share no data. This led to the decision to separate data from instructions so the independence property could be tracked. The design of a new simulation environment, *the General Simulation System (GSS)*, was launched in 1982. It called for a connectivity matrix to determine what processes shared what data. Then when allocating processes to processors, the connectivity matrix could be used to determine if a process can run concurrently with those already running.

Separating Data From Instructions - A New Paradigm

The separation of data from instructions is a new software paradigm that provides significant benefits. First, it allows capitalization on the concept of independence. By limiting access to specified data structures, models can be made independent. This leads to a decomposition of the simulation database into separate data structures - defined as *resources* in GSS. Instructions are grouped into sets of rules defined as *processes*. Resources and processes are grouped into elementary models. Elementary models are grouped into hierarchical models. This is illustrated in Figure 2, which contains a model of a local telephone system with PBXs connected to a local switch. The resources (data structures) are contained in the ovals, and the processes (instructions) are contained in the small rectangles. These can be edited directly as shown in the boxes.

In GSS, the interconnection of processes and resources is done graphically using icons and lines. This provides the ability to produce an engineering drawing of the architecture of a model, where lines connecting processes to resources determine what processes have access to which resources. Models can be connected to each other by connecting a process in one model to a resource in another. Independence of models can be inspected visually by looking at the number of lines connecting them.

Independent Instanced Models - Modeling Made Easy

The above concepts have led to the independent instanced model as part of the GSS environment. This allows a modeler to build a single model along physical lines, just like building a single piece of equipment. This model can then be instanced many times, automatically, in a simulation. This paradigm makes it easier to develop models on a large parallel processor than by using current methods on a single processor. This capability has been implemented as part of this Phase I contract, and is described in Chapter 7.

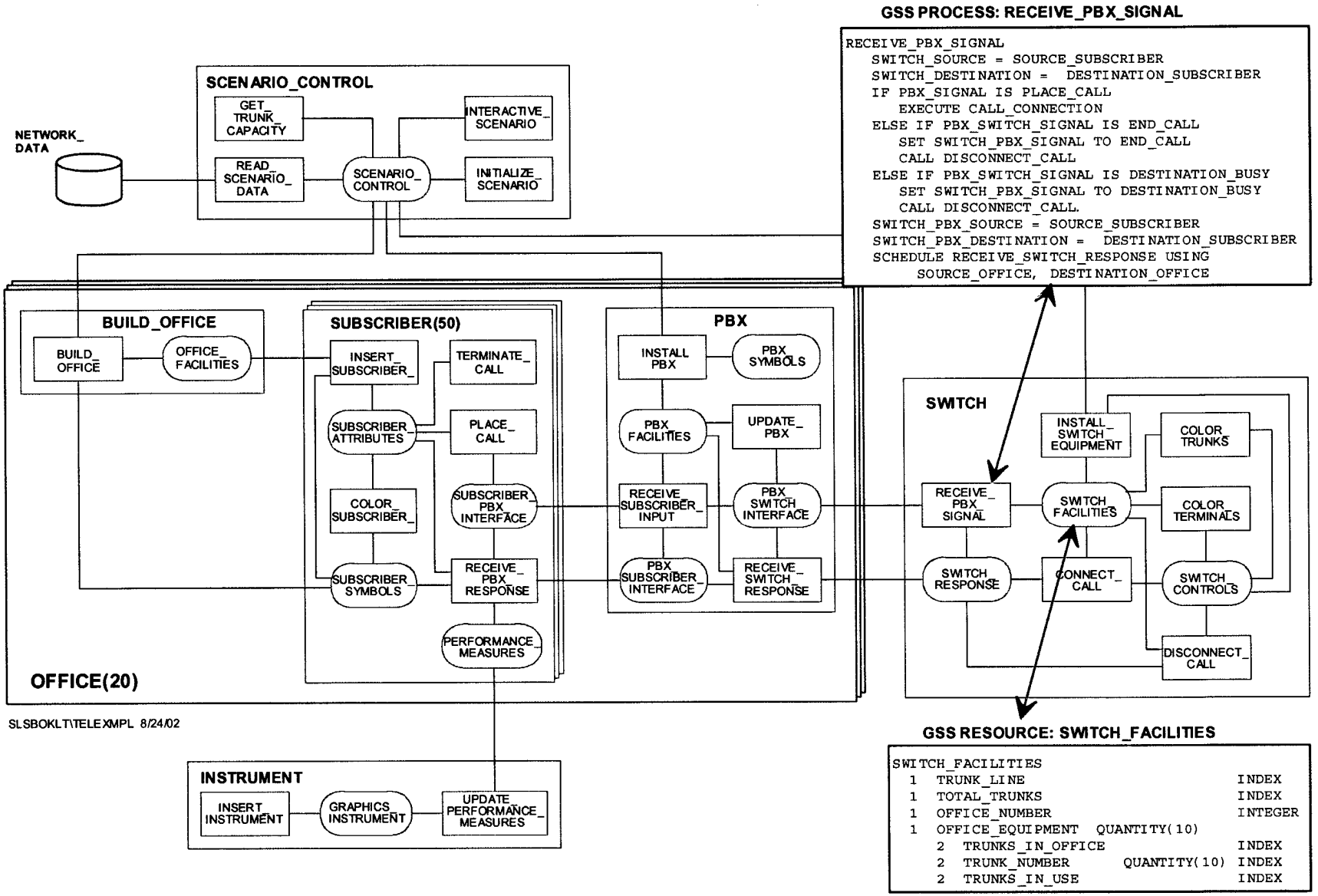


Figure 2. Telephone network models in GSS.

The Concept Of A Generalized State Vector

Separating data from instructions has clarified the meaning of the state of a model or simulation. It is defined by the state of all of the resources in that model or simulation. This has led to the concept of a *generalized state vector*. One can look at the state of a simulation as one big state vector comprised of all the resources in that simulation. Alternatively, a simulation is partitioned into a set of sub-states corresponding to the resources or subvectors.

This paradigm allows reuse of many concepts from the state space framework. For example, the simulation state vector as used in GSS is considered to represent a *generalized coordinate system*. It is up to the modeler to come up with the best set of states to make the problem easy to solve. This implies selecting the set of resources that simplify the transformations of state that represent the dynamics of the system. These transformations are embodied in the processes. When a process runs, it starts with the initial state of the attached resources and takes them to the next state.

This is the same problem as picking the best set of variables or coordinate system to simplify a set of partial differential equations. As indicated above, courses that cover problem solving in the applied sciences, e.g., physics and engineering, stress that choice of a coordinate system is the key to making a problem easy to solve. In GSS, one selects the best breakout of resources (state subvectors) to simplify the processes (transformations of state).

We have expanded that concept to the generalized state vector, one that consists of general information, not just variables that take on numeric values. Consider that a GSS resource is equivalent to a data vector containing states such as RED, YELLOW, and GREEN. The data can be English words or character strings, as well as numbers. A generalized state vector may consist of one or more subvectors, i.e., GSS resources.

With the above in mind, a GSS simulation consists of a very complex *simulation state vector* (the simulation's data base) that changes as processes are invoked. At any instant of time, the simulation state vector contains the values of all the resources in the simulation. It must also contain the simulation queue, the simulation clock, and the real-time clock and random number generator seed if used. External files are considered inputs to a nonhomogeneous model as described in [10], and are not part of the model's state vector.

State Space Definition Of A GSS Model

A GSS model only has access to a subset of the simulation state vector when a process in that model is running. This is the *model state vector*. A subset of the model state vector contains those resources that reside within the model. Resources residing in one model can be shared with another model. The state space representation of GSS is shown in Figure 3. The state vector that a model has access to consists of the following items and their corresponding information elements:

- ACCESSIBLE RESOURCES - The resources that processes within a model are attached to. Note:- These resources need not reside within the model.
- SIMULATION QUEUE - The entries in the queue, including the indices it uses when it's processes are scheduled.
- SIMULATION CLOCK - The time of the simulation clock, including priority, if and when it schedules another process.
- REAL TIME CLOCK - The value of the real-time clock if and when it uses the real time clock.
- RANDOM NUMBER GENERATOR - The value of the current random number generator seed if and when it uses the random number generator.

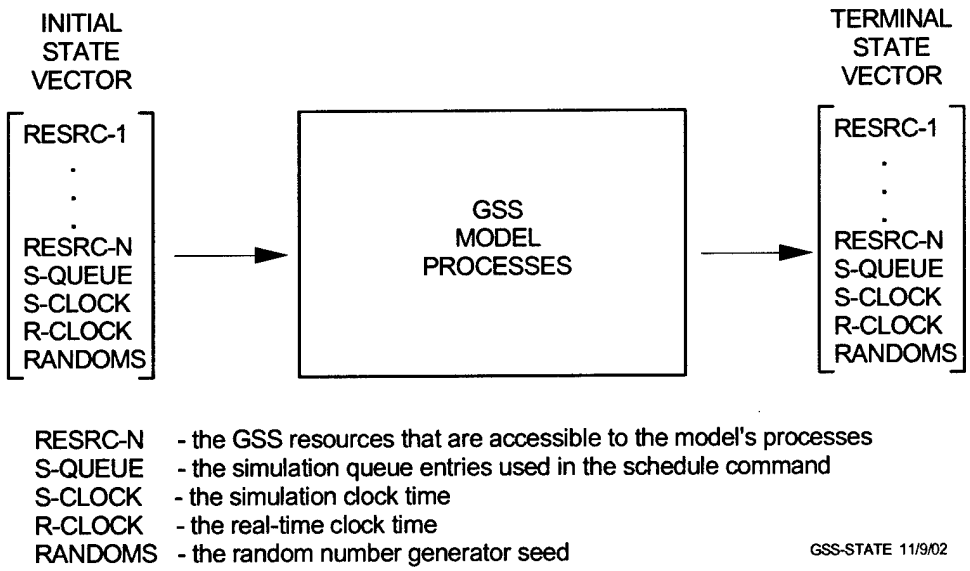


Figure 3. State space representation of GSS.

GSS processes are scheduled based upon the logic within itself or other processes. When a GSS process *runs*, it can schedule itself or other processes at specified times in the future, or at the current time. GSS processes run in zero *simulated* time. At any time, the state of a model depends solely upon its state vector. When a process in a model runs, its *terminal state*, i.e., the value of its substate vector - when it passes control back to GSS - depends solely upon its *initial state*, i.e., the initial value of its substate vector, and the rules within the process. When processes in another model share a part of the state vector of a given model, then any future state of the given model is, in general, dependent upon the rules in the other model, since they can change the given model's state vector.

Analogy To Symbolic Models Using State Space

The state space representation of a GSS model, Figure 3, is analogous to a set of equations that represent the state of a dynamic system at any instant in time. All future states are represented by the *equations of motion* in state space notation, and the initial conditions, reference Gelb, [7]. Electrical engineers have become accustomed to a graphical representation of the differential equations of electrical circuits, using interconnected icons of resistors, capacitors, inductors, generators, transistors, transmission lines, etc., refer to Figure 4. Such a drawing defines the differential equations of motion of the changes in electrical voltages and currents in the circuit. Given the initial conditions, the state of the circuit is defined for all time thereafter. In other words, the total dynamical description of the network is defined by the symbolic network.

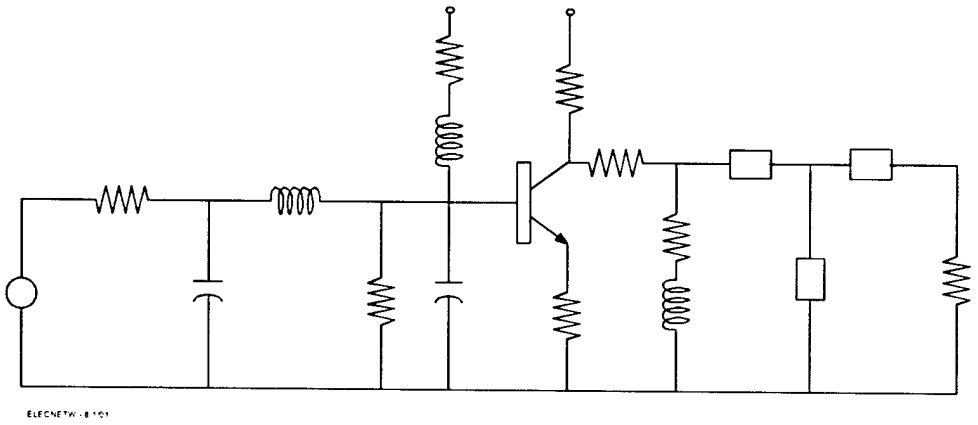


Figure 4. Iconic representation of an electrical network.

In GSS, the interconnection of resources and processes, as shown in Figure 2, is analogous to the electrical circuit drawing in Figure 4. Each has its corresponding *rules* and *storage* underlying each primitive element. In the case of electrical circuits, there are constituent equations that describe the changes in energy storage in differential form for each primitive icon. Representation of any system element must conform to this form of change.

In the case of GSS, sets of rules operate on sets of attributes (contained in resources) to define the elementary change relationships in a model. The engineering drawing shown in Figure 2 and the underlying rule and data structures, define the total state of the simulation at any point in time after the initial conditions. This is known as the *generalized state space framework.*

Choosing the Most Convenient Reference Frame

As described above, the generalized state space framework, as implemented in GSS, supports the representation of discrete event systems as well as discrete time and continuous systems. Figure 5 illustrates that generalized state space provides the underlying framework for representing dynamic systems.

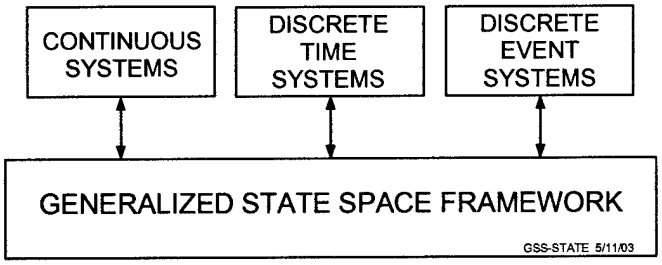


Figure 5. Generalized State Space framework for representing dynamic systems.

The difference between representations of a system's dynamics is a matter of convenience. A particular representation can be selected to support the economics of analyzing or predicting specific system behavior. If a system is conveniently represented by a set of differential or difference equations, then one of those representations might be best. If the system is more easily described by sets of rules operating on sets of attributes, then that representation should be chosen.

Since the advent of the digital computer, people have moved from analytical methods for integrating differential equations to numerical methods, especially when the systems represented are either nonlinear or nonstationary. Fast numerical algorithms for solving stiff nonlinear systems typically use complex heuristic approaches. What is interesting is that these approaches can be implemented more explicitly using GSS rule and attribute structures. As computers provide significantly greater memory and speed advantages, the space for solving problems is growing, alleviating restriction to numerical methods for solution, and moving rapidly toward heuristic rule based approaches using complex data structures. These approaches are compared in *Simulation Of Complex Systems*, [10].

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

Reusability Analogy

In the case of electrical circuit modeling, a transistor model may require a significant effort to build and validate. Once completed, that model can be shared in many different simulations, as well as hundreds of instances used in a given simulation. Similarly, for models built using GSS. Development and validation may require significant effort, whereupon a given model can be shared in many different simulations by different organizations, as well as appear in hundreds of instances in a given simulation.

Complex models of electrical elements, such as transistors and transmission lines, may be made up of the primitive elements, and represented by higher order symbols. One can *push down* on these symbols and bring up the primitive representations that show all the detail underlying the model. More complex networks, such as groups of digital circuits in the form of gates and flip-flops can be represented using another level of hierarchy. In this manner, complexity is pushed down to the level that one wants to see it, and removed from view when it only serves to cloud the picture. This aids in both the understandability and reusability of a model.

Similarly, one can represent complex models in GSS using a hierarchy of models, wherein higher level icons are used to represent the highest level of a model, and one can push down as many times as needed to get to the primitive layer. In GSS, the primitive layer consists of resources and processes. This also aids in model understanding and reusability.

An Alternative Approach To Generalized State Space

In 1987, Ramadge and Wonham, [11] and [12], described the need to use English words as states in a control system. They introduced the notion of *alphabets* to deal with these non-numeric states. Their finite-state machine approach is somewhat different than that of the generalized state vector, particularly in the implementation of models describing complex systems. However, it appears that the underlying effect of these two concepts is essentially the same. Although there are no journal publications on this method, the generalized state space approach is documented in copyrighted GSS User's Manuals and PSI books on model development going back to 1982 and 1983. It is believed that these approaches were conceived independently.

4. PHASE I TECHNICAL OBJECTIVES

In Phase I, PSI proposed to design and demonstrate a CAD tool that subject area experts/engineers can use directly to generate both their software and simulations. Our intent was to provide an interface that applications engineers relate to easily, rather than using expensive parallel computer programmers to translate their problems. Such a tool can generate efficient parallel computer code automatically if the user interface and resulting software are designed properly. Furthermore, if the application engineer wants to use a different MPP platform, his CAD tool interface should remain identical, generating the required software. This concept is illustrated in Figure 6.

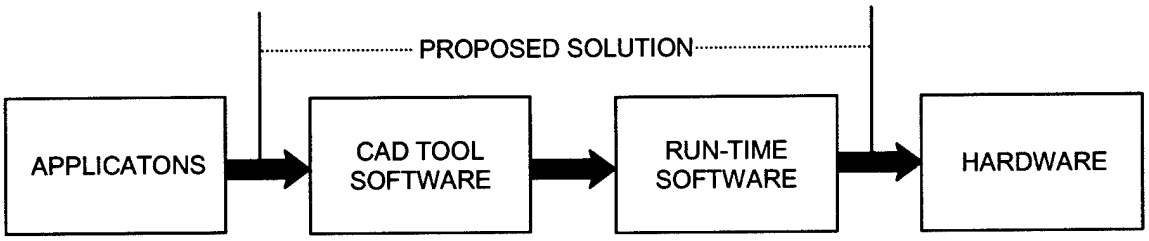


Figure 6. CAD tool approach - designing the software first.

Since PSI has already developed the visual CAD technology that: (1) provides users with a graphical architecture environment to design independent software modules; and (2) contains the architectural database needed to support the parallel processor run-time environment, the object of this research has been to analyze the architectural database so that it could be used to modify the run-time environment that optimizes the allocation of parallel processors to processes. Without information on the application software architecture, the allocation process is naive and effectively random.

The research performed in Phase I derived the pertinent statistics from our large software database so that they can be analyzed for pragmatic design decisions in preparation for Phase II. In Phase II, we will build and test candidate design approaches using actual systems and simulations, leading to a selected product for Phase III. Having already performed an analysis on this problem, we have determined that selection of a parallel processor technology for testing is an important part of this effort. This selection is addressed under our plan for Phase II.

5. OVERVIEW OF THE USER INTERFACE TECHNOLOGY

When PSI designed its CAD environment for discrete event simulation, the two major issues addressed were: (1) the difficulty of building valid models; and (2) the time to run a realistic scenario. The difficulty in building valid models was due to the complexity of the software. Run time may have been reduced by parallel processing, but the investment was huge and risky. To address these issues, PSI developed a CAD approach that leads directly to effective use of highly parallel processors. We note that software applications are considered easier to implement on a parallel computer than discrete event simulations because of the requirements to (1) synchronize each process with the main simulation clock; and (2) ensure synchronized data coherency to meet validity requirements. From this standpoint, we consider the software problem to be a subset of the simulation problem.

Separation Of Data From Instructions For Efficient Processor Allocation

In software, separating data from instructions violates the OOP rules. In the hardware world, this paradigm is not new. Data and instructions are separately stored and managed on today's chips. PSI considers this an essential software paradigm for effective use of parallel computers, where one has to allocate processes to processors efficiently. This implies knowing which processes can run concurrently, which implies that they must be independent. Independence is effectively determined by whether or not they share data. If allocation is to be done automatically, the allocation manager must have the information on who shares what data. PSI's technology is built upon this concept. The most significant paradigm shift in our development environment is the separation of data from instructions.

The resulting properties of our technology provide enormous benefits. First is the ability to represent software graphically, with a one-to-one mapping from the drawing to the code. Second is that software architectures can be designed and reviewed from an engineering standpoint to determine module independence. Third is the resulting connectivity map of what processes share what data. Fourth is what processes reside inside what modules. If modules are independent, then processes within those modules are best migrated to the same processor. This information is stored in our run-time as well as development databases. It is this information that provides our ability to optimize the allocation of processes to processors to maximize run-time efficiency. These benefits are best described by an example.

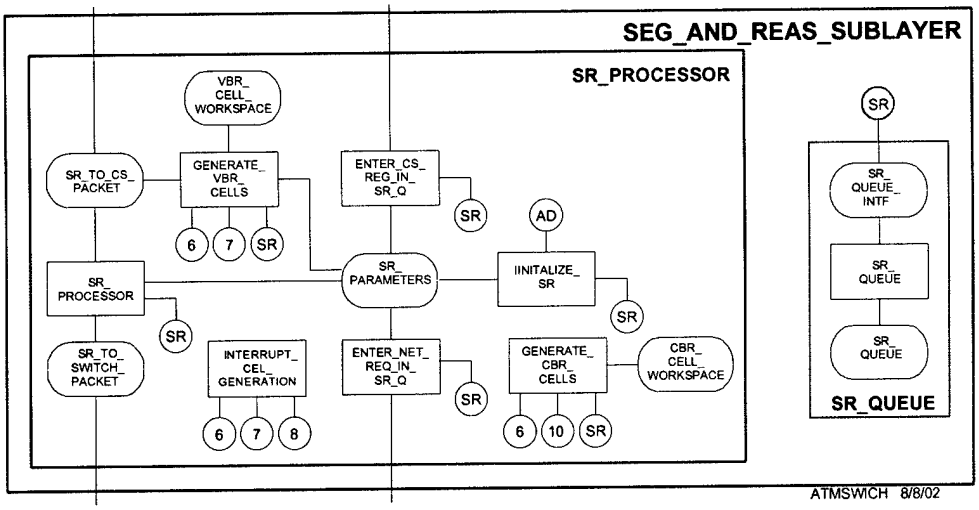


Figure 8. SEG_&_REAS_SUBLAYER module.

```
RESOURCE NAME:  MESSAGE_FORMATS

STANDARD_MESSAGE
 1      SYNC_CODE                                  CHARACTER 5
            ALIAS   VALID_CODE   VALUE  '10101',
                                        '01010'
 1      TYPE                                       STATUS FORMAT_A
                                                          FORMAT_B
 1      CONTENT                                    CHARACTER 46

FORMAT_A          REDEFINES STANDARD_MESSAGE
 1      PAD                                        CHARACTER 13
 1      HEADER_A
        2       MESSAGE_PRIORITY                   STATUS FLASH
                                                          PRIORITY
                                                          ROUTINE
        2       ORIGIN                             INDEX
        2       DESTINATION                        INDEX
                    ALIAS  BROADCAST  VALUE 0
 1      BODY_A
        2       BODY_LENGTH                        INTEGER
 1      TRAILER_A
        2       MESSAGE_NUMBER                     INTEGER
        2       TIME_SENT                          REAL
        2       TIME_RECEIVED                      REAL
        2       ACKNOWLEDGMENT                     STATUS RECEIVED
                                                          NOT_RECEIVED
        2       LAST_SYMBOL                        CHARACTER 2
                    ALIAS  TERMINATOR   VALUE '\\', '//', '<<','>>'

FORMAT_B              REDEFINES STANDARD_MESSAGE
 1      PAD                                        CHARACTER 13
 1      HEADER_B
        2       SOURCE                             INDEX
        2       SINK                               INDEX
 1      BODY_B
        2       CONTENTS                           CHARACTER 42
```

Figure 9. A resource - a hierarchical data structure.

```
PROCESS:              SR_PROCESSOR

RESOURCES:      SR_TO_CS_PACKET                          INSTANCES:    NODE
                SR_PARAMETERS
                SR_QUEUE_INTF
                SR_TO_SWITCH_PACKET
```

```
PKT_SR_PROCESSOR
    EXECUTE GET_SR_MESSAGE
    EXECUTE PROCESS_SR_MESSAGE
    IF QUEUE_STATE IS NOT EMPTY
        SCHEDULE SR_PROCESSOR
          IN PROCESSING_TIME MICROSECONDS USING NODE
    ELSE SET PROCESSOR_STATUS(NODE) TO IDLE.

GET_SR_MESSAGE
    SET  SR_QUEUE_INTF REQUEST TO DEPART
    CALL SR_QUEUE USING NODE

PROCESS_SR_MESSAGE
    IF PACKET_TYPE IS A CELL
        EXECUTE PROCESS_CELL
    ELSE IF PACKET_TYPE IS A REQUEST
        EXECUTE PROCESS_REQUEST
    ELSE EXECUTE INVALID_PACKET_TYPE.

PROCESS_CELL
    MOVE SR_QUEUE_INTF MESSAGE TO SR_TO_SWITCH_PACKET
    IF SR_TO_SWITCH_PACKET DESTINATION IS EQUAL TO NODE
        EXECUTE CHECK_PAYLOAD_DEST
    ELSE IF SR_TO_SWITCH_PACKET SOURCE IS EQUAL TO NODE
        EXECUTE CHECK_PAYLOAD_SOURCE
    ELSE EXECUTE INCORRECT_NODE.

CHECK_PAYLOAD_SOURCE
    IF PAYLOAD_TYPE IS USER_VOICE
        EXECUTE PROCESS_CELL_VOICE_SRC
    ELSE IF PAYLOAD_TYPE IS USER_DATA
        EXECUTE PROCESS_CELL_DATA_SRC.

CHECK_PAYLOAD_DEST
    IF PAYLOAD_TYPE IS USR_VOICE
        EXECUTE PROCESS_CELL_VOICE_DEST
    ELSE IF PAYLOAD_TYPE IS USER_DATA
        EXECUTE PROCESS_CELL_DATA_DEST.

PROCESS_CELL_DATA_SRC
    EXECUTE GET_MESSAGE_INFO_CELL
    INCREMENT MESSAGE CELLS_TRANSMITTED
    EXECUTE UPDATE_MESSAGE_INFO
    IF MESSAGE CELLS_TRANSMITTED IS EQUAL TO MESSAGE CELLS_TO_TRANSMIT
        EXECUTE GET_NEXT_MESSAGE.
    CALL ENTER_USER_REQ_IN_VC_Q USING NODE


    ...          (This process is incomplete - 2 additional pages are not
shown!)
```

Figure 10. A process - a hierarchical set of rules.

To insure independence of modules, PSI has developed a set of architectural design rules that can be enforced automatically as the designer builds modules graphically. This involves viewing a module as an *N-port module* as used in electronics hardware design. By limiting the number of lines (wires) at a port to two, the independence of modules is ensured. Note that we have not considered any aspects of coding, which in VSE or GSS is confined to the language environment. We have only analyzed the module architecture - graphically! These design rules assure ease of module understandability and independence, and therefore real reuse. They are the major reasons we have been able to build and validate the world's largest simulations at very low cost. This same technology is ideally suited to make effective use of highly scalable parallel processor computers.

Another departure from typical software is the integrated management environment of VSE and GSS that completely tracks the architecture behind the scenes, and contains the databases to determine both spatial and temporal independence at run-time. Modules are tracked through all of the hierarchical levels needed by the designer to control design complexity. Every resource and process is tracked relative to what processes have access to what resources within multiple module instances. This database can be used to adaptively manage the allocation of parallel processor resources during run-time based upon knowledge of module instance independence at any level in the hierarchy. Load balancing can be achieved concurrently through selected instance migration. This critical information is not available anywhere else!

We will now relate the number of module instances to opportunities for parallelism. As the top level modules, e.g., a switch, take on higher degrees of complexity, they become significant opportunities for highly efficient parallel processing. If the switch is modeled along physical lines, its physical counterpart operates concurrently with its neighbors. Therefore, independent module instances in a simulation can also run concurrently in a parallel processing environment. Such instances are not limited to simulation, but exist frequently in real-time control and communication systems.

Based upon this concept, our hypothesis is as follows: As the number of instances of a complex independent module increases, the number of parallel processors that can be used effectively increases proportionately, just due to the independent module instances. Similar opportunities for effective use of processors can also be obtained within a top level module instance, down to the process level. This is because of the hierarchical design and resulting scope of a VSE or GSS process.

For example, the ATM_TRANSCEIVER within the ATM_SWITCH in Figure 7 can have 20 instances (one for each port), all tied to the same instance of a switch. A scenario of 100 switch instances can invoke a total of 2000 ATM_TRANSCEIVERs. We can envision many instances of subscribers as well as other packet and circuit switches running concurrently, interfacing with each other through links or gateways. Each of these instances can run concurrently since almost all of the processes and resources are *interior* to the instance and therefore *independent* of the other instances.

Quantifying The Importance Of Software Architecture.

To better understand this typical architectural phenomenon, consider the modules in Figure 11.

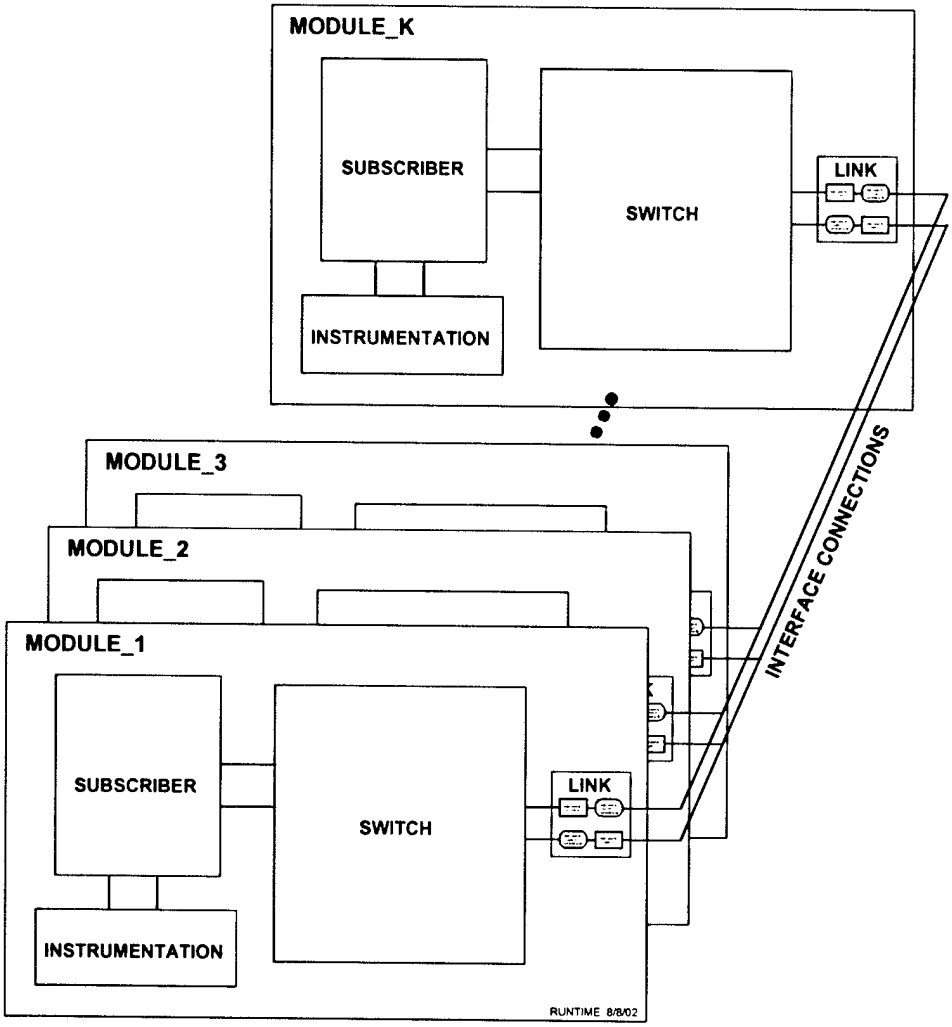


Figure 11. Independent *instanced* modules connected by an interface.

The top level modules in Figure 11 are drawn alike for simplicity, but in fact may be different types or instances of the same type. As an example, we will consider an MSS simulation with 100 circuit switches, 50 packet switches, and 50 ATM switches. Consider that each instance of each switch is part of a single module along with its corresponding subscriber submodule instance that generates and receives voice calls and data messages and files, and its instrumentation submodule that takes measures of traffic. These large submodules are the largest part of each module. A link interface submodule also exists connecting each top level module. Except for the two processes connected from each module to the interface, all other processes in each module are independent of those in any other module, i.e., they share no other resources between modules. This is done by design - *of the software architecture.*

6. OVERVIEW OF THE RUN-TIME TECHNOLOGY

Significant work has been done by PSI on prior projects for the Army as well as DARPA toward development of the required run-time technology. This work covers the use of the module architecture knowledge described above as well as knowledge of the independence of individual processes at the module boundaries to determine what processes can run concurrently. This work includes development of the protocols required to ensure data coherency of resources shared across module boundaries and used by processes in different processors. It includes the synchronization of scheduled processes running on separate processors in a simulation. It provides for controlled variations in synchronization that ensure validity of results of a simulation - something not provided by other approaches, e.g., the Time-Warped Operating System, and its derivatives (e.g., SPEEDES). It provides for optimal ordering and scheduling of p-threads.

Figure 12 below provides a top level view of the proposed design for the VSE/GSS run-time environment for an MPP environment. In addition to the Process Scheduler, there is a Processor Allocator to allocate processes scheduled at the current time (or within a pre-defined ΔTmax in a simulation) to the available processors. We plan to use standard OS level calls to assign parallel threads (p-threads) to processors. This will provide the ability to allocate specific processes to specific processors.

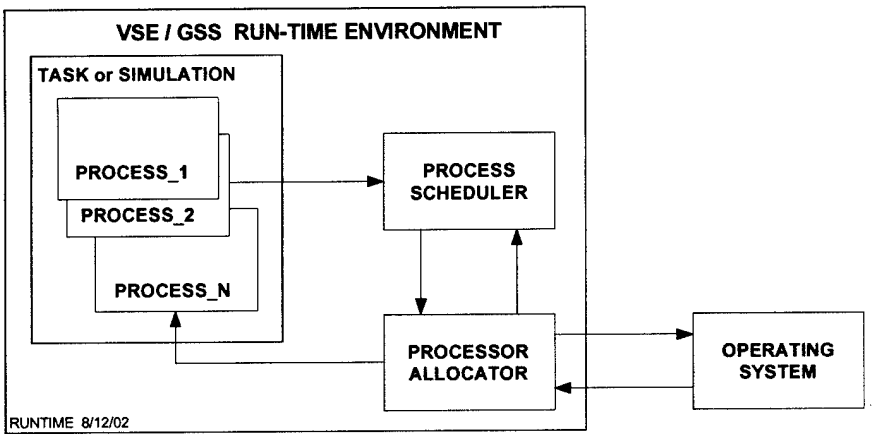


Figure 12. Connection between the VSE process scheduler and the processor allocator.

There are additional mechanics of this environment to be characterized, e.g., the nature of the dynamic changes to the schedule versus the state at time T. This will affect the algorithm design for optimal ordering in minimal time. Instanced modules create special submatrices of the connectivity matrix that are independent. These become candidates for quasi-independent queue management, potentially in separate processors. PSI's work in discrete event simulation for the past 20 years has provided us with significant knowledge of solutions to these types of problems. In addition, processor load balancing must be considered in more detail, but this has been the subject of much prior research, both at PSI and elsewhere. Finally, marrying this new technology to hardware must be started in the architectural design stages. We have worked with many hardware vendors in the past, and are prepared to work with them again.

Summarizing The Importance Of The Software Environment

Given applications with a high degree of inherent parallelism and very efficient parallel computers, their effective use comes down to three major factors. First is ensuring that full advantage can be taken of the inherent application parallelism - a software design problem. Second is balancing the load - a run-time software problem. By separating data from instructions and using the visual development environment that PSI has already developed, we have the software architectural knowledge to do both well. The third and most important factor is making it easy for the subject area experts to describe their problem, without having to twist it into a special computer language. PSI's success in CAD tools for building very complex discrete event simulations and software tools has already demonstrated the ease with which this is done. We feel confident in our ability to bring large scale parallel processing power into the mainstream of computing via *ease of use* - the winning "WinTel" approach.

7. INSTANCED MODELS

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

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

DEFINITIONS

Classes Of Models And Their Elements

- INTERIOR AND INTERFACE ELEMENTS - Processes (resources) are *interior* elements of an elementary model if they have no shared interfaces with resources (processes) outside that model. They are *interface* elements if they do have such shared interfaces. The interior elements of an elementary model are interior to any higher level model containing the elementary model.

- INTERIOR AND INTERFACE MODELS - Models are *interior* to a hierarchical model if they contain no elements with shared interfaces outside that hierarchical model. They are *interface* models of that hierarchical model if they do have elements with such shared interfaces. Models that are interior at a given level of a hierarchy are interior to all higher levels.

- INSTANCED RESOURCES - Resources are defined to have multiple *instances* when they are elements of an instanced model at the architectural level. This implies that, at run time, *each instance* of that resource exists as an *independent copy* of that resource and is referenced by a unique name determined from the resource name and instance number.

- INSTANCED MODELS - Models can be defined to have multiple *instances* at the architectural level. This implies that, at run time, *each instance* of the model must have an *independent* copy of every resource in the model, corresponding to *instanced resources.* When invoked at run time, processes contained in an instanced model are assigned instance numbers that reference their corresponding resource instances. Processes in one instance cannot share resources in another instance.

- HIERARCHICAL INSTANCED MODELS - Instanced models may be defined within instanced models hierarchically. Resources contained in the lowest level instanced model will have as many independent copies as the product of the successive instances in the hierarchy. These will be referenced by a unique name determined from the resource name and successive instance numbers. When invoked at run time, processes contained inside the lowest level instanced model are assigned instance numbers that reference the corresponding hierarchy of resource instances.

- SHARED INTERFACES - Models that are connected by shared resources have *shared interfaces*. For example, two models have a shared interface if a process inside one model shares a resource with a process inside another model. The shared resource is the shared interface.

Classes Of Independence

- SPATIAL INDEPENDENCE - Two processes are *spatially (memory) independent* if they share no resources (memory), independent of time. Two models are spatially independent if every process in one is independent of every process in the other, i.e., they have no shared interfaces. Interior processes of an instanced model are spatially independent from those of other instances of the same model. Model instances are spatially independent if they have no shared interfaces. Interior model instances are spatially independent. From here on *independent* will imply spatially independent.

- TEMPORAL INDEPENDENCE - Processes (models) can be *independent in a given instance of time,* but dependent in another instance of time. If two processes are using the same instance-pointer value to reference a resource in an instanced model at the same time, then they are not independent at that time. However, if at another time they reference mutually exclusive instance-pointer values for that same resource, they are independent.

Classes Of Schedulers

- SYNCHRONIZED SCHEDULERS - Spatially independent processes can run concurrently in the same time instance, independent of time, and are thus candidates for local scheduling on separate processors by schedulers synchronized by time instances, but otherwise independent.

- Processes interior to a model or model instance can be scheduled by a synchronized scheduler provided they are not scheduled with another scheduler *and* do not share resources with a process scheduled with another scheduler.

- MASTER SCHEDULER - Processes that are only temporally independent must be scheduled by a master scheduler since they are not spatially independent.

GENERAL RULES AND CASES

As described in the final report *Visual Software Development For Parallel Machines,* [13], most of the opportunities for inherent parallelism occur with instanced models. This is based upon the assumption that *model instances are independent,* i.e., except for the possibility of one or two processes at the boundary, processes in one instance cannot share resources in other instances. Along with these opportunities come considerations for automating the design approach to instanced models. Before describing the detailed design for interprocessor time and space (memory) synchronization, we will investigate the architecture of instanced models and their interconnections.

Figure 13 illustrates many of the design issues to be considered relative to using instanced models. If RECEIVER in RADIO schedules R_F_LINK in R_F_LINK, it can specify the SOURCE and DEST instances and thereby expect R_F_LINK to be tied to the correct instance of LINK_INFORMATION. Since the instance of RECEIVER is known when the schedule statement is invoked, the instance of the TRANSCEIVER resource can be passed implicitly to RF_LINK as well.

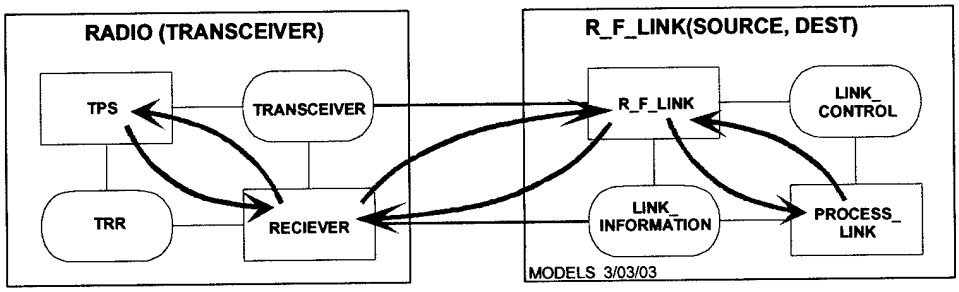


Figure 13. Example of an architecture with instanced resources.

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

If, however, TPS schedules RECEIVER, there is no way to know *automatically from the architecture* what instance of LINK_INFORMATION the RECEIVER process should be connected to. Therefore, this connection, shown in red, cannot be allowed with this type of call or schedule statement.

Figure 14 presents a similar case when KP in MODEL_3 schedules LP in MODEL_2. In this case, there is no way to pass on the pointer *automatically* to resource TRS in MODEL_1. This implies that, if a resource is to be shared between MODEL_1 and MODEL_2 when LP is called from outside MODEL_1, that resource must reside within MODEL_2.

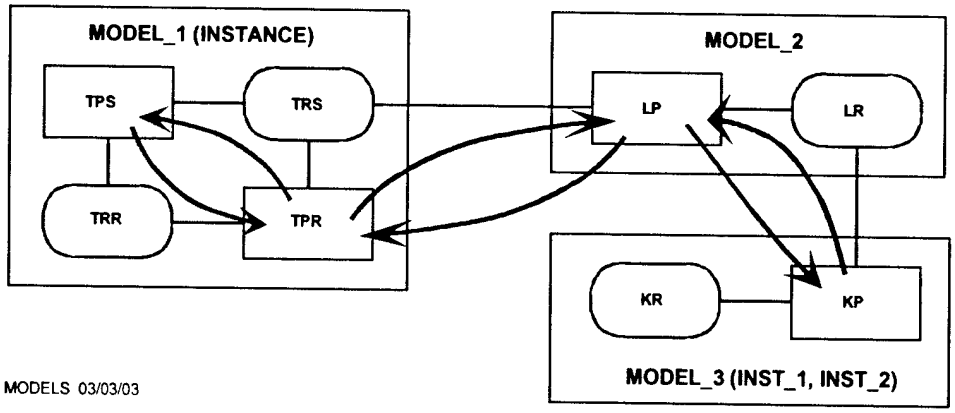


Figure 14. Example of another architecture with instanced resources.

There appears to be a generic rule that applies to Figures 13 & 14. When a process connected to an instanced resource is scheduled, the instance pointers for that resource must be specified *automatically from the architecture*, i.e., *explicitly* via the instance pointers in the schedule statement or *implicitly* based upon residence within an instanced model. In the case of the instance pointers, they must match an instanced model containing the resource; else the connection cannot be made automatically.

CALCULATING RADIO CONNECTIVITY

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

Figure 15 uses a radio model as an example. Each radio can have links to many others. A radio can only operate properly on one link at a time. However, the receiver model must account for the potential interference coming from other radios that are transmitting at the same time. Therefore, each radio must be connected to an environment model that provides for all of the possible cross-link connections between radios.

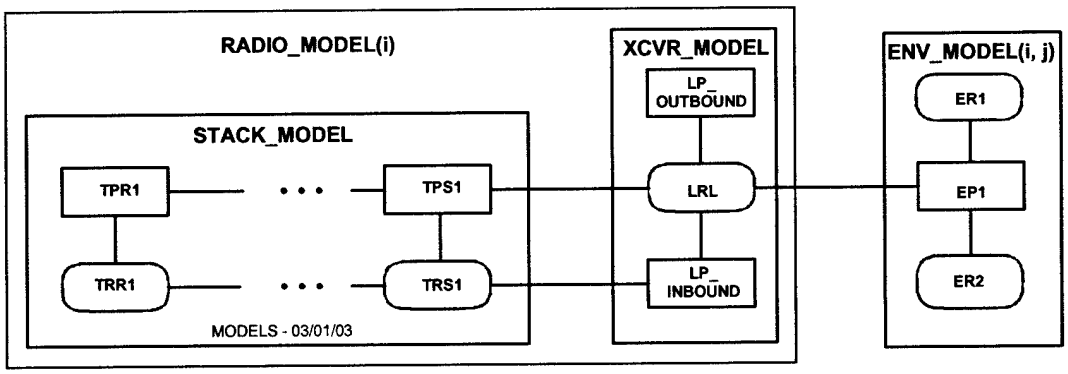


Figure 15. Example of good architecture for instanced resources.

The radio model in Figure 15 has an instance(i) for each radio. The environment model has an instance for each destination receiver(j) coupled with each source transmitter(i). All environment link instances (i, j) may operate concurrently, just as each radio can operate concurrently. In the case of collision analysis, i.e., when two or more transmitters transmit to more than one receiver at the same time, it is necessary for the model to have access to all link information at the same time. When an instanced model interfaces with a noninstanced model, the noninstanced model can present a bottleneck that, depending upon the architecture, can be significant.

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

INSURING MODEL INSTANCE INDEPENDENCE

For model instances to be independent, processes in one instance must not share any resources in another instance. Except for the interface resources, this is true for the architecture in Figure 15. With this architecture, each radio model instance can reside on a separate processor. Likewise, each environment model instance can reside on a separate processor. It may be better that environment model instances reside on the same processor as the corresponding radio model instances to minimize the time to move data between processors. This is shown in Figure 16. This may be a trade-off between operating in parallel and operating sequentially. However, message transfer implies a degree of sequential processing between corresponding instances of affiliated models, and this architecture may also be best for a single processor.

Consider that radio instance(1) transmits a message to radio instance(2&4). This is accomplished by having instance(1) of process LP_OUTBOUND scheduled with the message to go out. Since LP_OUTBOUND only interfaces with resources that are interior to RADIO_MODEL(1), the instance pointer, 1, is passed implicitly. In this case, the message is placed in LRL(1).

LP_OUTBOUND then schedules EP11 as the transmitter for radio 1. EP11 uses instance LRL(1) to get the message as well as ER11. This allows process EP11 to transfer data from LRL(1) to the selected receiver resources ER21 and ER31, and schedule EP21 and EP31 on computers 2 and 3.

On computer 2, EP21 schedules LP_INBOUND to receive the message, passing the pointer to ER21 where the message is currently stored. LP_INBOUND then takes the message from ER21 and places it in LRL. Similarly, on computer 3, EP31 schedules LP_INBOUND to receive the message, passing the pointer to ER21

This sequence of events represents what typically occurs in a real communication system where most of the events are occurring concurrently with other events. We note that the transfer of messages from radio(i) to radio(j) can be sequential. However, many pairs of radios can be doing similar transfers concurrently, and this is where the inherent parallelism exists. This parallelism is best realized in a simulation if the model architecture follows the same physical design as the architecture of the real system.

In the case that one must process information from all instances concurrently, e.g., when doing calculations based upon signals from every radio, then a utility model can be called from the environment model. This utility can store information on every link. It becomes a potential bottleneck in that it may be used by every instance. However, it may be used infrequently, typically only when there is movement or power changes in radios. This implies that, on the average, links operate independently, occasionally requiring cross-link calculations for all links.

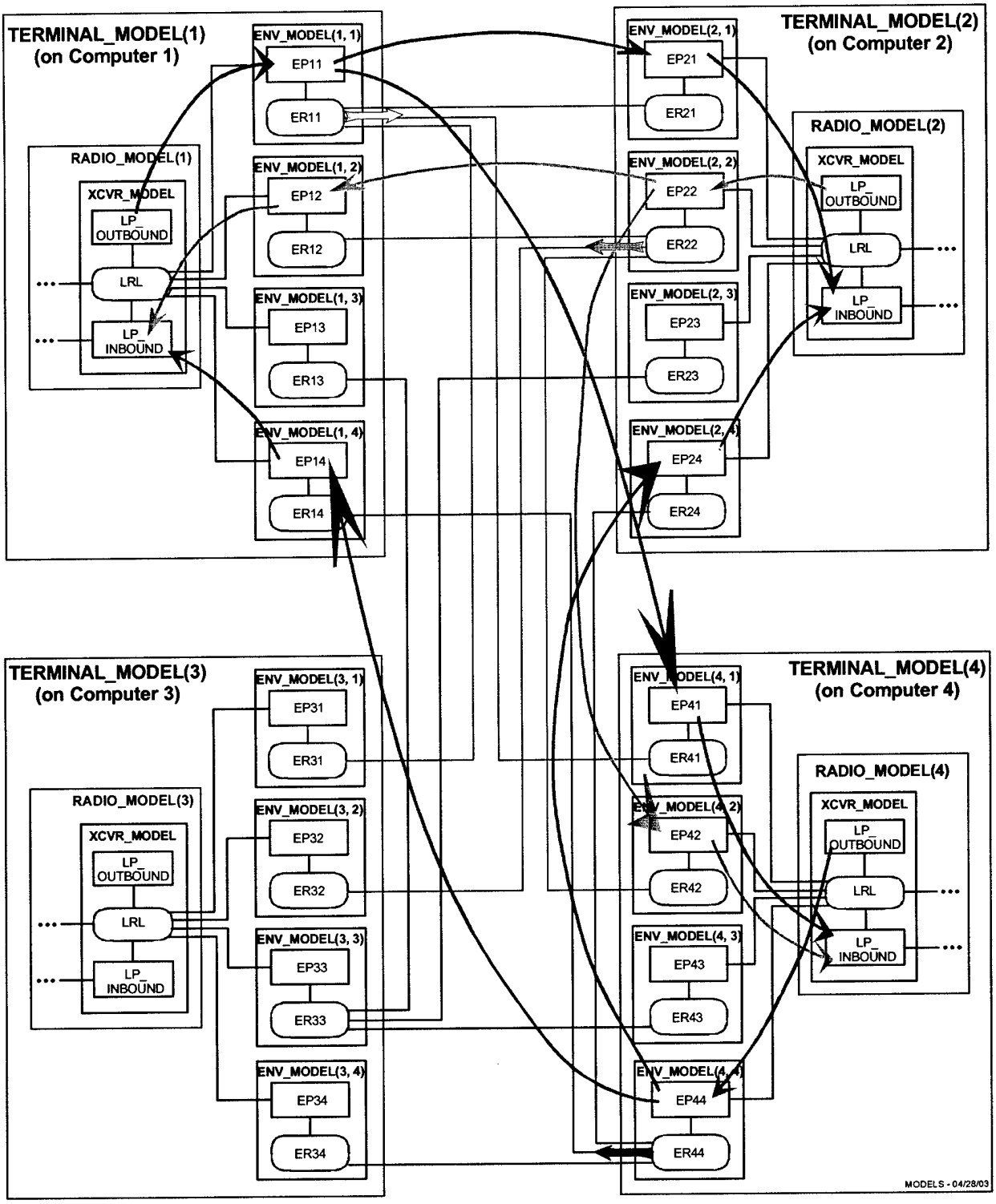


Figure 16. Example of a parallel architecture for instanced resources.

INSTANCE POINTER VALUE RULES

To specify an instance from outside an instanced model, the instance values (up to a maximum of 6) are assigned by the model in a schedule or cancel statement. The general format for a schedule statement is as follows.

SCHEDULE process_name INSTANCE instance_pointer_1, ..., instance_pointer_n

When a process starts to execute, the instance pointers defined for models containing that process hold the current values of the instances that the process represents. These instance pointers are used to automatically attach the proper resource instances to the process when it runs. Instance pointers are also available for use by the process in a *read-only* mode, i.e., values of model instance pointers cannot be changed by processes within that model instance. When one process is scheduled by another in the same model instance, the instance pointers are passed implicitly and *must not appear in the argument list.* Because model instances must be independent, processes in an instance cannot schedule any in a different instance of the same model.

Referring to Figure 15 above, TPS1 can SCHEDULE LP_OUTBOUND with the pointers back to the proper instance of TRS1 being automatically invoked. If LP_OUTBOUND schedules EP1 in ENV_MODEL(i, j), then it must explicitly use the form:

SCHEDULE EP1 INSTANCE *source, dest,*

where source, dest can be any properly defined numeric attributes or literals. Note that trying to connect LP_OUTBOUND directly to ER1 would not be permitted architecturally, since there is no way for LP_OUTBOUND to attach to the proper instance of ER1 when scheduled by TPS1. This follows from the independence properties. EP1 can schedule LP_INBOUND.

CALL STATEMENT RULE

CALL statements are sequential; they cannot be used to increase the number of concurrent processes (parallel paths). They directly control any processes they invoke at the time, rendering them nonindependent from the calling process. Calls from instanced models will automatically carry the current value of the instance pointer to the called process. If independent models are to be run concurrently, they must be scheduled. Calls may serve to invoke a process on another processor, e.g., a utility, but this may not be an efficient way to use the processors containing either the process or the call statement. Multiple copies of frequently called utilities are a more effective solution if they can run in parallel. This represents the typical time memory tradeoff, with memory being relatively inexpensive today. This leads to the desire for utilities that can be copied (or instanced) to be distinguished from those that can't.

MODEL INSTANCE CASES OF CONCERN

Case 1 SCHEDULEs, CANCELs & CALLs from a noninstanced model to an instanced model.

Referenced model (process) instances must be identified by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer.

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

References to the instance pointers of processes within the same instance are implicit, being resolved automatically by the process translator and run-time monitor. Values of the instance pointers of a model are read-only by processes within that model, and cannot be changed by them.

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

References across instances of the same model must be accomplished by using a shared interface in a separate model. Direct references are not permitted across different instances of the same model.

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

The instance pointer of the referencing process is passed automatically to the referenced process by the run-time monitor, without any explicit reference to point back to the resource instances in the referencing model that the referenced process shares with it.

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

The modeler must identify the referenced process instance by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer. Again, pointers to resource instances within the referencing model that are shared with the referenced model are automatically passed to the referenced process.

	CASES	SINGLE PROCESSOR	PARALLEL PROCESSORS
1	SCHEDULEs, CANCELs & CALLs from a noninstanced model to an instanced model	Referenced model (process) instances must be identified by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer.	Referenced model (process) instances must be identified by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer.
2a	SCHEDULEs, CANCELs, & CALLs within the same instance	References to the instance pointers of processes within the same instance are implicit, being resolved automatically by the process translator and run-time monitor. Values of the instance pointers of a model are read-only by processes within that model.	References to the instance pointers of processes within the same instance are implicit, being resolved automatically by the process translator and run-time monitor. Values of the instance pointers of a model are read-only by processes within that model.
2b	SCHEDULEs, CANCELs & CALLs across instances of the same model	References across instances of the same model must be accomplished by using a shared interface in a separate model. Direct references are not permitted across different instances of the same model.	References across instances of the same model must be accomplished by using a shared interface in a separate model. Direct references are not permitted across different instances of the same model.
3a	SCHEDULEs, CANCELs, & CALLs from an instanced model to a noninstanced model.	The instance pointer of the referencing process is passed automatically to the referenced process by the run-time monitor, without any explicit reference to point back to the resource instances in the referencing model that the referenced process shares with it.	The instance pointer of the referencing process is passed automatically to the referenced process by the run-time monitor, without any explicit reference to point back to the resource instances in the referencing model that the referenced process shares with it.
3b	SCHEDULEs CANCELs & CALLs from one instanced model to another instanced model.	The modeler must identify the referenced process instance by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer. Again, pointers to resource instances within the referencing model that are shared with the referenced model are automatically passed to the referenced process.	The modeler must identify the referenced process instance by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer. Again, pointers to resource instances within the referencing model that are shared with the referenced model are automatically passed to the referenced process.

GENERAL RULES

- When a process in an instanced model is scheduled or called, the instance pointers must be specified explicitly if not implicitly. The values of the pointers are set as follows:
  - When referenced from a process outside the model, the model instance must be specified as an instance_pointer after the process name.
    Example: SCHEDULE process_name INSTANCE instance_pointer
  - When referenced from a process inside the same model instance, the instance pointer must not appear in the instance_pointer list.

- When a process within an instanced model references another process in that same instance, it automatically invokes the same instance pointers. No arguments are specified relative to the common model instances after the process name.

- References to hierarchical model or other multiple instance pointers must be ordered as specified in the instance pointer list of the process being called. This must be in the order of the hierarchy, from the top down, *with instance pointers that do not reference model instances going last.*

- If a process within a hierarchically instanced model is scheduled from outside a subset of the instances, only the new instance pointers must appear in the instance pointer list of the process, in order from the top of the hierarchy down.

- Reuse of instance-pointer names in resources attached to any process interior to an instanced model must be qualified.

CREATING AND ADDRESSING INSTANCED MODEL RESOURCES

Since the quantity clause will not be used explicitly to create multiple copies of the instanced resources inside instanced models, these will be created automatically by the GSS translators and monitors. These will be created as instanced data structures, each with their own names, e.g., RES9901[m], RES9962[m,n], etc. Subscripted C pointers will be equated to these structure instances during model initialization by assignment statements as follows:

```
S9901(0001)  =  &RES9901(0001)
S9901(0002)  =  &RES9901(0002)
S9901(0003)  =  &RES9901(0003)
     .
     .
     .
```

Actual Pointer values

```
S9962(0001, 0001)  =  &RES9962(0001,0001
S9962(0001, 0002)  =  &RES9962(0001,0002
S9962(0001, 0003)  =  &RES9962(0001,0003
S9962(0002, 0001)  =  &RES9962(0002,0001
S9962(0002, 0002)  =  &RES9962(0002,0002
     .
     .
     .
```

Actual Pointer values

Up to six levels of model instancing are allowed, including any QUANTITY levels of hierarchy within the lowest level model. These C pointers can then be passed via C function calls as follows.

```
next_process_function(S9901(instance_pointer_1),
                      S9962(instance_pointer_2, instance_pointer_3))
```

8. INTERPROCESSOR COMMUNICATIONS

Under the Phase I contract, multi-processor communication protocols were built and tested at the user's simulation level. This was accomplished using the Inter-Processor (IP) Resource developed by PSI. The goal of this step was to provide an easy-to-use facility for inter-processor communications. In the final implementation, these resources will be used for communications between multiple processors, each running separate model instances of a simulation, and each communicating with the other.

In the Phase I experiment/demonstration, information was sent between GSS user level processes on different processors. These processes were connected to GSS interprocessor resources that automatically provided the link between the processors. During this experiment, models on one processor sent messages to models on the others while data was collected to verify that concurrent processing and communication were operating successfully.

8.1 IP RESOURCE COHERENCY MANAGER

The IP Resource Coherency Manager provides a coherent communications link when processes share a resource between two or more processors. New protocols were developed to insure coherency of multiple copies of the same resource residing on separate processors. As shown in Figure 17, GSS resources can be shared by processes on different processors as IP resources.

This architecture is required for both networked computers and an MPP. The requirement is for coherency of data shared between processors. If the processors are in networked computers - implying separate operating systems - then additional levels of protocol are needed to communicate. If the processors are in an MPP, then the protocol must match the memory transfer mechanism across processors, e.g., the KSR machine had automatic cache coherency.

To support coherency, an interlock mechanism is required to ensure that, when a process sharing an IP resource is running, a process on another processor sharing the same resource cannot run concurrently. In addition, if a process has just used an IP resource on one processor, any process sharing that resource on another processor must receive the latest copy before it can run.

Finally, there are two types of protocols needed to support fast processing of simulation models at the interface between many processors. These are the following:

- **Asynchronous** - The latest copy of an IP resource (the one connected to the process that ran last) gets passed to the next process that needs it in another processor. Data coherency is guaranteed (independent of time).
- **Synchronous** - Everyone gets the latest copy from a single source, a one to many interface. This depends upon time synchronization (by design) to ensure data coherency.

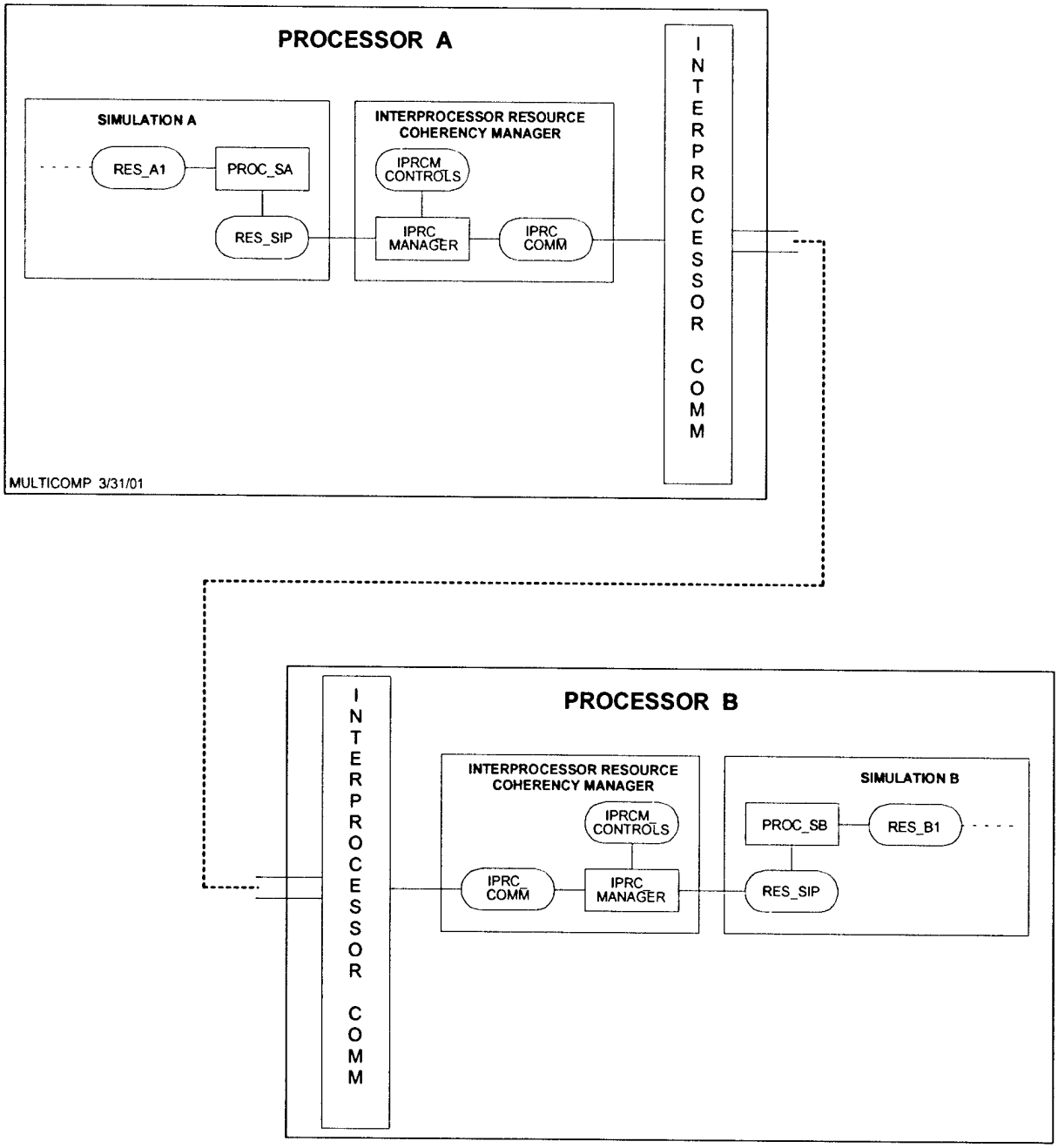


Figure 17. Resource coherency manager architecture.

The desired protocol must be specified by the designer when running in a parallel processing environment. It is specified relative to the IP resource, i.e., an IP resource must be shared using either the synchronous or asynchronous protocol. In the case of multiple computers, matching IP resources must use the same protocol. These protocols are further specified below.

Asynchronous Coherency Protocol

When the Asynchronous Coherency Protocol is used, the latest copy of an IP resource (the one connected to the last process to run) gets passed to the next process that needs it in another processor. When a process (e.g., P_A) that is connected to an IP resource gets scheduled, the coherency protocol is invoked. This protocol first checks to see if P_A controls the IP resource, i.e., it is the last process using the resource. If so, it has the latest copy already. If it is not the controlling process, it must request the latest copy from the controlling process. If more than one process requests a copy, they are queued up on a first-in first-out FIFO basis. Data coherency is guaranteed (independent of time).

The following functional rules are implemented in the asynchronous coherency protocol used for an MPP environment. The rules for a multiple computer simulation environment are slightly different.

- One copy of an IP resource must reside on each processor that contains a process that shares it.
- When a process that shares an IP resource is running, processes sharing other copies cannot run. The process that is running has control of the IP resource until it passes control to a process that shares it on another processor.
- Before a process that shares an IP resource can run, a check must be made to determine if that process has control. If that process does not have control, the IP resource manager must initiate a request to its counterpart on the processor containing the controlling process. It must then wait to receive a return signal. If the signal contains passage of control along with the latest copy of the resource, the resource is controlled (locked) for use only by that requesting process.
- If control has been transferred before the request is received by the prior controlling process, the request is sent on to the processor that has control. A FIFO queue will be built of processes that have requested the IP resource. The queue is passed from processor to processor with the IP resource.
- At some point, we may make a decision to attempt to schedule the next process in the queue, or to wait for control of that resource to be passed to the requesting process. This decision must consider that the resource in use may be shared by a large number of model instances, implying a high level of contention.

In addition to building the IP resource coherency manager to implement these rules, modifications were made to the development monitor, process translator, control specification translator, and run-time environment. These modifications account for recognition of IP resources, and processing of the master and slave control specifications so that tables are built to determine which processes share IP resources, and what machines they reside upon.

These facilities were designed to provide a clear speed advantage since (1) the GSS environment is a tool that needs to be tailored only once on a given platform; (2) the protocols used are generally transportable; and (3) speed is the predominate reason for using parallel processing. This is not an area where speed was sacrificed for simplicity of the software.

8.2 DEVELOPMENT MONITOR MODIFICATIONS

The IP resource is a new entity in GSS. This new resource type was incorporated into the databases, lists, prompts, and decision processes. Changes were made specifically with regard to the prompts for determining the resource type attached to a given process. This resource is similar to the inter-task resource, and the modifications followed along the similar lines. In addition, all query reports that list resources by type were modified.

9. INTERPROCESSOR SCHEDULING

9.1 MASTER AND SLAVE SYNCHRONIZERS

Before a simulation can run on multiple processors, whether SMP or networked, knowledge must reside on each processor regarding the processes that can be scheduled on each of the other processors. In a networked computer environment running multiple simulations, this requires exchanging data among the processors used in the simulation to support scheduling processes in another simulation - on another processor.

Figure 18 illustrates this requirement. Note that the simulation on processor A schedules processes on processors B and C as well as itself. The simulation on processor C only schedules processes on itself and processor D. When a process is scheduled in a different processor, this schedule request must be sent to that host processor. To accommodate these cross-schedules, the scheduling mechanism on each processor must have knowledge of the host processor that each of its scheduled processes resides upon. This information must be available before each simulation starts.

To meet these needs, a *process assignment* database is built containing the pertinent cross-processor scheduling information corresponding to each simulation. This database is then attached to each simulation at run-time to support the distribution of cross-processor schedule requests to the proper processor. This is illustrated in Figure 18. This database contains the name of each simulation control specification, and the processor's host-id, for each process that is cross-scheduled by the simulation running on that processor, as well as each process in that simulation that can be scheduled. It is derived from information on each process in that simulation. The data structure is shown below.

```
IP_SCHEDULE_TABLE    QUANTITY(1000)
    1  PROCESS_NAME                 CHAR 24   *** CROSS-SCHEDULED PROCESS
    1  HOST_PROCESSOR_ID            CHAR 24   *** CONTROLLING PROCESSOR ID
    1  PATH_SIMULATION_NAME         CHAR 60   *** SIMULATION ID
```

The master simulation control specification names the host-IDs of the subordinate simulations. It also requires that the control specifications of the subordinate simulations have been prepared, and that the process assignment database file has been created for each before preparing the master simulation control specification. Then, when the master simulation control specification is prepared, all of the subordinate simulation process assignment databases are accessed, all of the cross-processor schedule statements are reconciled with respect to the processes scheduled, and the subordinate simulation databases are updated with the host-IDs for each process that is cross-scheduled.

Figure 18 shows the master and slave synchronizers. The master synchronizer tracks the earliest scheduled time of processes in the ΔTmax interval. It determines when the clock can be set to the next interval, and handles cross-schedules going both ways on processor A. The synchronizer modules on the other machines report their earliest schedule time, to the master, and when they have reached the end of their ΔTmax intervals.

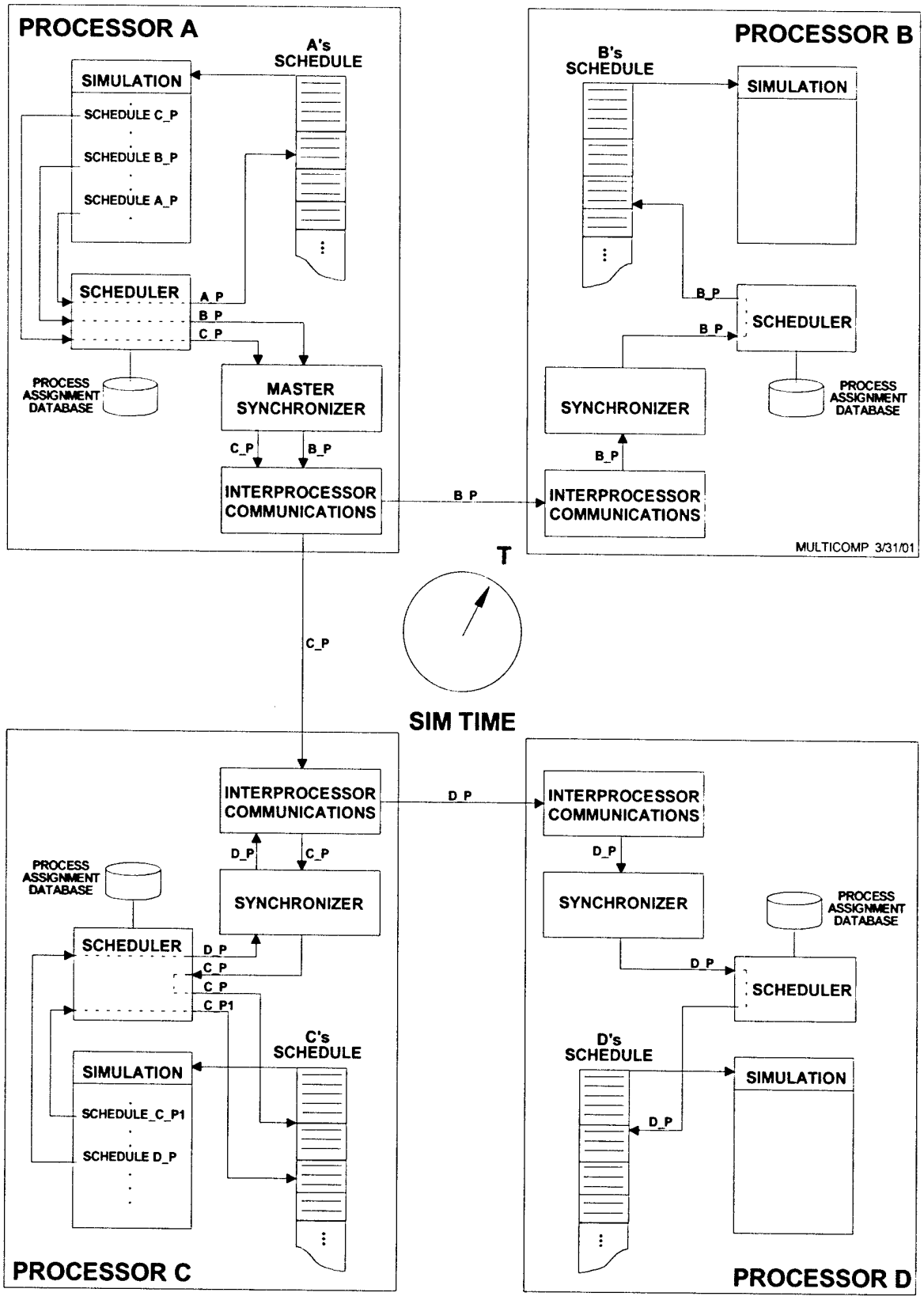


Figure 18. Cross-schedules for a four processor case.

9.2 PROCESS TRANSLATOR MODIFICATIONS

The process translator was modified to accommodate the new cross-processor schedule statement defined below. Code is now generated to call on the new synchronizer module to pass the resulting schedule requests to the proper processor. This code contains a pointer to the record in the IP_SCHEDULE_TABLE corresponding to the process being scheduled. The value for this pointer is determined during translation of the subordinate simulation control specification. The HOST_PROCESSOR_ID and PATH_SIMULATION_NAME are determined by the master simulation control specification translator. The IP_SCHEDULE_TABLE is read during initialization of each simulation. Then the code generated by the process translator calls the synchronizer, passing the pointer to the proper record in this table for cross-scheduling.

Cross-Processor SCHEDULE And CANCEL Statements

New SCHEDULE and CANCEL statements in GSS provide for cross-processor scheduling and canceling of processes. These statements add an "IN ... SIMULATION" clause to qualify the process_name as follows.

Format
{SCHEDULE / CANCEL} process_name IN simulation_name SIMULATION [[...]]

9.3 CONTROL SPECIFICATION TRANSLATOR MODIFICATIONS

In addition to the modifications required for supporting the IP resource coherency manager, the control specification can contain a statement for specifying the ΔTmax interval defined below. The new translator also provides for initialization of the databases to be used for cross-processor scheduling. These databases are created by the master simulation control specification translator reading files produced by all of the subordinate simulation control specification translators, and then building a file for each of the subordinates as well as itself for initialization at run-time. These changes are described in more detail below.

When a cross-processor schedule statement is translated, code is generated to call the synchronizer to initiate a schedule request to the processor containing that process. The synchronizer then performs the table look-up to determine the processor to which the request must be sent.

Specifying The ΔTmax Interval

The "DELTA_TIME = " statement specifies the ΔTmax interval for that portion of the hierarchy controlled by this particular simulation control specification. The format for this statement is shown below.

Format
DELTA_TIME = Δtmax_interval [time_units]

The time_units options are given in Appendix 3 of the GSS User's Reference Manual. They range from PICOSECONDS to DAYS.

If the time_units option is not used, time is assumed to be in seconds (the default).

If a DELTA_TIME statement does not appear in the list, it is assumed to be 0 (the default).

Synchronizer Database Initialization

As described above, commands to schedule processes residing in another simulation at run-time are communicated via InterProcessor (IP) communications to a PATH_ SIMULATION_NAME within a HOST_PROCESSOR_ID. To accomplish this, each simulation reads these names during initialization (one set for each process cross-scheduled in the simulation) from its corresponding process assignment database, and stores them in a table available to the synchronizer. To create this database, each of the subordinate simulation control specification translators produces their initial process assignment databases. These initial databases contain the names of all of the processes in that simulation, as well as the names of the cross-scheduled processes and their simulation names. These are put into files that the master simulation control specification translator reads to reconcile all of the cross-scheduled processes within each of the simulations. It then updates these files with the corresponding HOST_PROCESSOR_IDs, and ships them back to the hosts/paths for each simulation so they can load their IP_SCHEDULE_ TABLEs during run-time initialization .

File Processing Summary

During translation, each control specification translator creates a list of the processes *cross-scheduled* by that simulation. It also creates a list of all the processes that *reside* within that simulation, since they can be scheduled by a simulation on another processor. Both of these lists are available to the master control specification translator. These files, one for each simulation, are updated by the master translator with the host-id of each simulation. They are then used by each of the respective simulations during run-time initialization.

9.4 SCHEDULER MODIFICATIONS

Schedule Synchronization

When a process that is currently running on a given processor terminates, the next process is retrieved from the schedule queue. In the single processor case, the clock can simply be advanced at this time, and the process invoked. In the multi-processor case, if the clock advances beyond the Te+ΔTmax interval, then it must wait until all processors have reached the same condition. When this happens, a new Te+ΔTmax interval is set, and the checks are made again. Any processor with a process in the Te+ΔTmax interval can proceed to invoke that process. Others must wait for the proper interval.

When the simulation clock in any processor exceeds the Te+ΔTmax interval, a notification is sent to the master synchronizer containing the processor/simulationID. In addition, all cross-schedule requests are sent to the master synchronizer before being sent to the processor containing the cross-scheduled process. This latter information is used to update the status maintained in the master scheduler regarding the number of processes (if any) to schedule beyond the Te+ΔTmax interval. For example, if processor A sends a signal to the master synchronizer that its simulation clock has exceeded the interval, but a cross-schedule is sent to that processor subsequently, it is not finished. However, the order of presentation will insure that the master simulation clock will advance beyond the Te+ΔTmax interval only after the clock-time of the next process to be scheduled in every simulation is beyond the interval.

9.5 SYNCHRONIZER DESIGN

When a process passes the interval test, i.e., it falls within the interval, it is checked to determine if it shares an interprocessor resource. If it does, then the interprocessor resource coherency check is made. If that processes shares an interprocessor resource, then the process must wait until control of that resource is obtained by that process. When these conditions are satisfied, the process is invoked to run. The interprocessor resource coherency facility is described in the previous section.

The Te+ΔTmax interval is computed by the master synchronizer after receiving the next schedule time from each simulation and determining the earliest. After the current interval has completed, i.e., there are no more processes in any of the queues whose schedule times fall within the current interval, then the master synchronizer notifies all of the other simulations of the new Te value, signalling the start of a new interval. The synchronizer module shown in Figure 18 handles these communications, as well as the cross-processor schedules.

10. EXPERIMENT

In addition to the instanced model development effort described in this report, and as part of the final deliverable on this contract, PSI has built a multi-computer experiment on a networked cluster of twelve Intel computers. This experiment consists of the multi-computer version of the General Simulation System and the Netted Full Spectrum Sensor (NFSS) Operations Management System (OMS) that PSI has been building for the Army at CECOM, Ft. Monmouth. The OMS provides for the management of all of the Army's sensors, including the MASINT sensors. It involves taking in product reports from individual sensor control systems and fusing a picture of the battlefield.

The purpose of these experiments was multi-fold. It provided a test-bed for refining the data coherency, cross-scheduling, and synchronization algorithms that were designed and built on a prototype basis under previous efforts. It also provided for the collection of test data relevant to analyzing the effects of latency on the broad class of simulations that we classify as partially independent. This classification is described in the paper that has resulted from the experimental efforts reported upon here. This paper is attached as Appendix A to this report.

10.1 HARDWARE PLATFORM

The hardware platform used in the parallel NFSS experiment consists of twelve Intel computers networked with a Gigabit Ethernet switch as shown in Figure 19. The twelve machines are arranged in three sets of four. Each of the sets is represented with one keyboard, monitor and mouse via a KVM switch. All of the computers have access to shared directories that reside on computers 1 and 8.

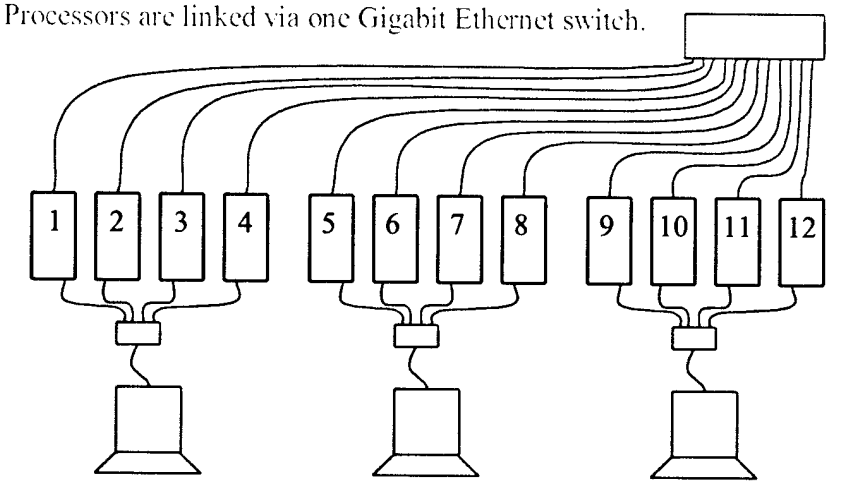


Figure 19. Cluster architecture.

The Intel computers used in the cluster are described in Figure 20. Since they all have identical hardware, Windows 2000 and supporting software was installed manually on one of the machines, which was then cloned onto the remaining eleven. This process resulted in a cluster with very similarly configured and performing machines.

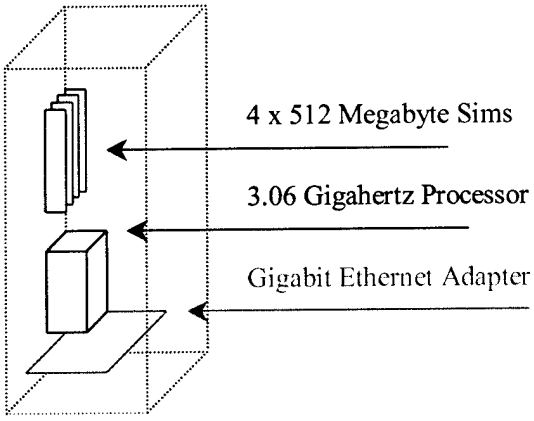


Figure 20. Individual machine configuration.

The Ethernet Adapters support Full and Half Duplex modes at 10, 100 and 1000 Mbps. The ability to *pin* the adapters at slower speeds was crucial for gathering communications latency type data.

10.2 THE GSS MULTI-COMPUTER SIMULATION FACILITY

As described throughout this report, the GSS Multi-Computer Simulation Facility is designed to work on multiple networked computers, each running their own Operating System (OS). The facility is designed to be independent of the number of computers. The version used for this experiment requires that simulations run on each machine (there can be more than one simulation on a machine) without load balancing.

Each machine contains a copy of the run-time environment consisting of the three subsystems. The Resource Coherency subsystem ensures that only one process is allowed to access a resource at a time and that processes have the most current version of the resources they share before they run. The Inter-processor Scheduling subsystem is used to invoke processes in other participating simulations in different machines, as well as the simulation that invokes the schedule. The Time Synchronization System is used to ensure that the clock drift between simulations is kept under a user specified ΔTmax. These systems will be expanded in future versions of the GSS parallel processing environment.

The cluster based parallel processing system used here communicates via TCP/IP. The communications architecture design omits a central hub, which would present a bottleneck. As the number of simulations increases, the TCP/IP infrastructure that is established to support it becomes significantly more complex. Most off this infrastructure is not required on single operating system machine architectures. Furthermore, the communications fabric used to support multi-processor, single operating system machines is as much as 60 times faster than Gigabit Ethernet.

10.3 NFSS-OMS SIMULATION OVERVIEW

The NFSS OMS system contains an embedded simulation of the detailed target environment, electromagnetic environment, sensors, communications, as well as actual multi-level/multi-sensor fusion algorithms and control systems to predict the responses to multiple sensor tasking. In addition to other challenges, this represents a multi-faceted high stress computational problem. We are using this real world system as the basis for analyzing and evaluating approaches to parallel processing. As will become evident in the Analysis of Results section below, the NFSS is also a good choice for analysis purposes because of the ease with which the sensor load across the cluster and the degree of inherent parallelism can be varied.

As shown in Figure 21, the NFSS OMS system includes three basic components. First, the emitter component represents transmitter ground truth. This component contains the transmit process, which is responsible for providing the sensor component with transmission locations. The sensor component includes the sensing processes that determine whether or not a given transmission was detected. This information is then furnished to the OMS component, which fuses the results together into useful reports.

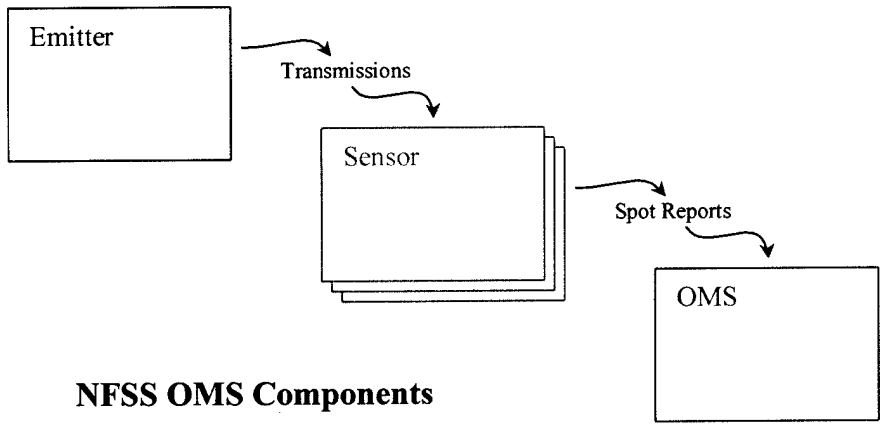


Figure 21. NFSS OMS components.

Figure 21 shows an instanced sensor component because many types of sensors can participate in the system. In this experiment, twelve similar sensor simulations were used (one per processor). The emitter and OMS simulations were run on a single processor with one of the sensor simulations because their loads are negligible compared to that of the sensors'.

The experimental NFSS also includes a graphics toggle that is used to turn graphics on and off. While graphics are very useful for debugging and documenting, they sometimes produce noise when used in timed runs. Figure 22 shows a scenario that includes two sensor types and enabled graphics. The top left window labeled *TRANS* shows the locations of the transmitters. As shown in the sensor simulation windows labeled *UGS_01* and *UGS_02*, the location of a transmission is depicted with a star in the sensor simulations. When graphics are enabled, lines are drawn from each of the sensors to the transmission location and colored according to whether or not the transmission can be detected. Finally, the OMS simulation labeled *DISP* registers a spot report that includes fused *found/missed* information from the sensor simulations. As illustrated in Figure 22, the transmission shown was detected by two of the sixteen sensors.

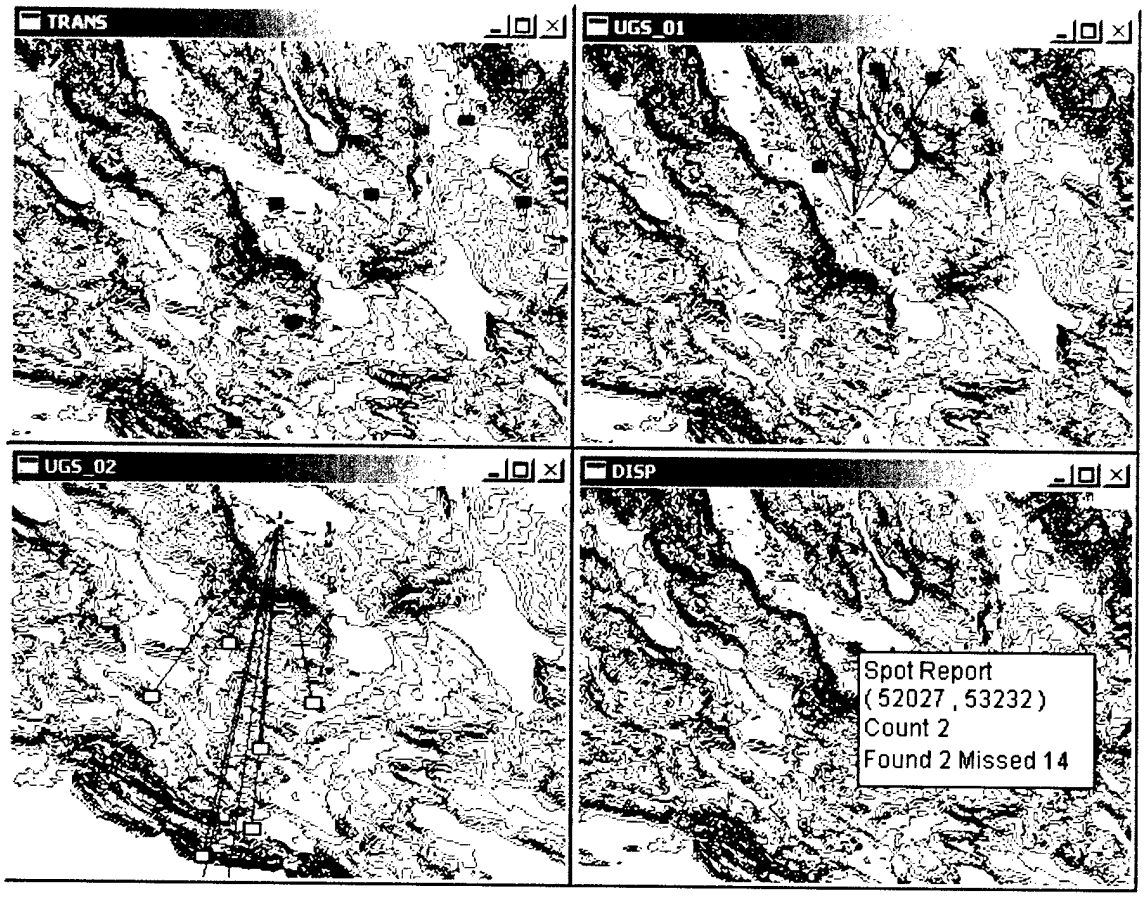


Figure 22. NFSS multi thread illustration.

In order to obtain real single processor runtime for use in efficiency calculations, the NFSS was altered to be able to run as a single thread. Figure 23 shows a single thread version of the above scenario. The single thread version was also used as a validity benchmark.

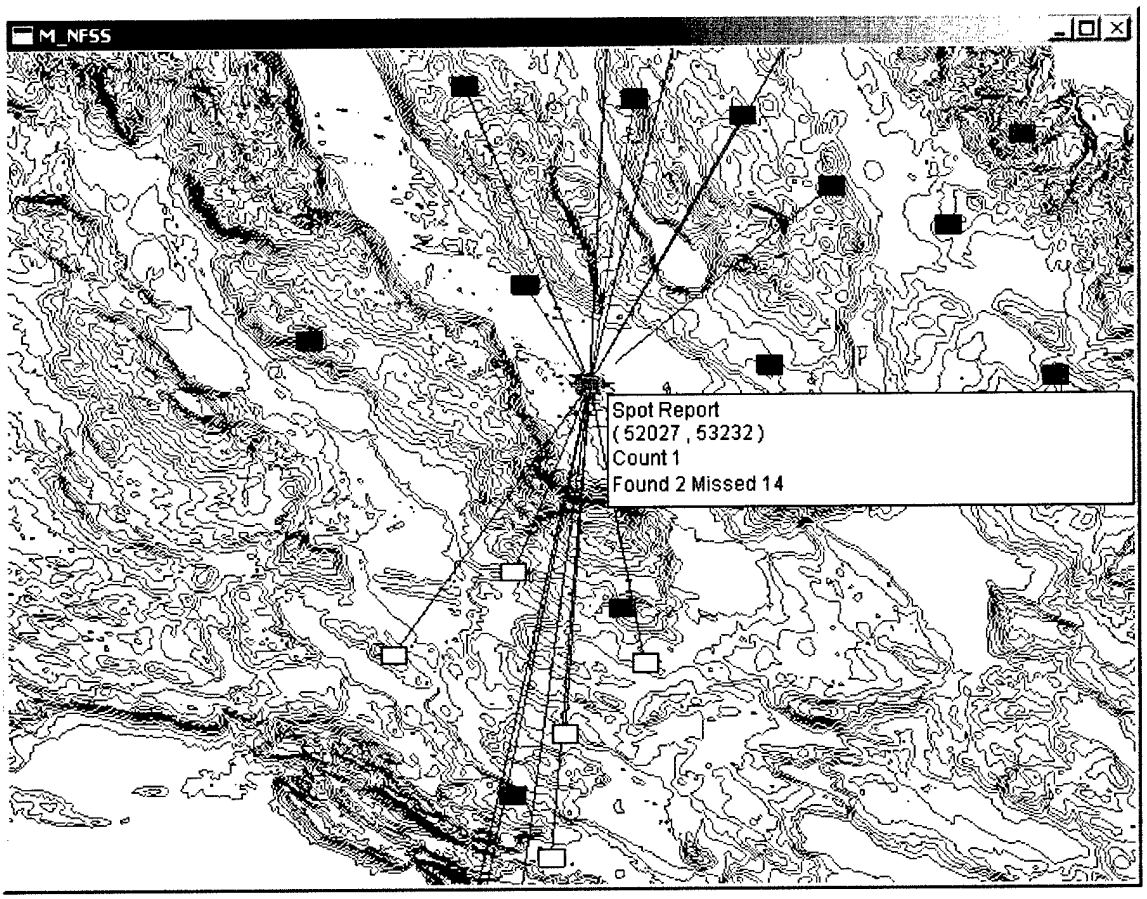
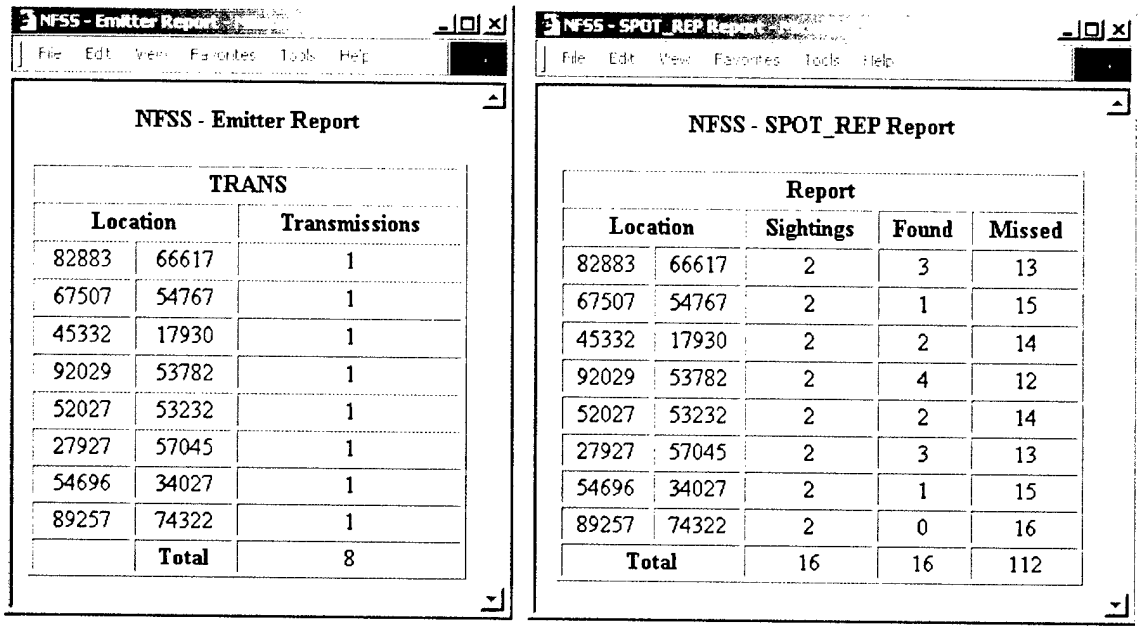


Figure 23. NFSS single thread illustration.

The NFSS produces two results reports for use in validation. Shown in Figure 24, the Emitter Report is generated by the emitter component and includes the location and number of transmissions for each transmitter. The SPOT_REP Report is generated by the OMS component and includes locations, number of sightings and *found/missed* information for each spot report.



NFSS - Emitter Report

TRANS		
Location		**Transmissions**
82883	66617	1
67507	54767	1
45332	17930	1
92029	53782	1
52027	53232	1
27927	57045	1
54696	34027	1
89257	74322	1
	Total	8

NFSS - SPOT_REP Report

Report				
Location		**Sightings**	**Found**	**Missed**
82883	66617	2	3	13
67507	54767	2	1	15
45332	17930	2	2	14
92029	53782	2	4	12
52027	53232	2	2	14
27927	57045	2	3	13
54696	34027	2	1	15
89257	74322	2	0	16
Total		16	16	112

Figure 24. NFSS generated reports.

10.4 THEORY

The NFSS as well as many other discrete event simulations uses a time synchronized architecture as illustrated in Figure 25. In this figure, wall clock time (T_{WC}) is shown in the negative y direction and is displayed in the first column. The second column shows time slot spans. Simulation processes and their simulation times are shown in the last column. Longer blocks in the last column represent processes that take more CPU cycles to run.

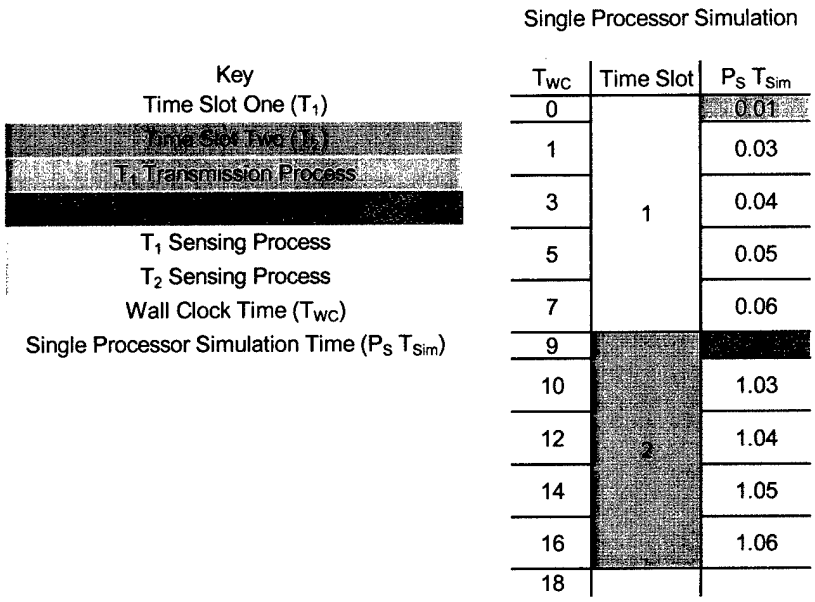


Figure 25. Single processor event sequence illustration.

At the beginning of the time slots illustrated in Figure 26, transmissions occur. Transmission processes are responsible for making transmission locations available to sensing processes. Sensing processes use transmission locations along with their own locations and the electro-magnetic environment model to determine whether or not transmissions are detected. It follows that the order of the sensor processes within a time slot is not important as long as they follow the proper transmission process. For instance, if a T_1 sensing process runs after the T_2 transmission process, the sensing process will use the wrong transmission location and therefore produce invalid results. In this way, the validity of the results is dependent on the sequential integrity of the time-step.

Three Processors ($\Delta T_{Max} < 0.01$)

T_{WC}	Time Slot	P_1 T_{Sim}	P_2 T_{Sim}	P_3 T_{Sim}
0		0.01		
1			0.03	
3	1			0.04
5			0.05	
7				0.06
9				
10			1.03	
12				1.04
14			1.05	
16				1.06
18				

Figure 26. Multi-processor event sequence with small ΔT_{max}.

When discrete event simulations are spread across multiple threads, it is possible that the individual simulation times may vary at any given wall clock time. In order to maintain validity, individual simulation clocks need to be synchronized.

Figure 27 shows the event sequence of the above scenario spread over three processors. In this figure, the maximum difference in simulation time (ΔT_{max}) is kept under 0.01 second. While this ensures the exact same event sequence as the single thread version, it precludes any of the processes from running at the same time. The white blocks in the figure indicate blocks of time where the processor is idle. Since no processes are allowed to run concurrently, the three processor version of the simulation cannot be faster than the single thread version and therefore the efficiency cannot be better than 1 over the number of processor used.

As mentioned earlier, the integrity of the time slot is maintained as long as its sensing processes run after its transmission process and before the next time slot's transmission process. Figure 27 shows the event sequence of the above scenario across three processors with a ΔT_{max} greater than 0.01 second and less than 0.2 seconds. This ΔT_{max} is large enough to permit sensing processes to run concurrently and small enough to ensure that they run after the intended transmission process. Efficiency is improved while validity is maintained.

Three Processors ($0.01 < \Delta T_{Max} < 0.2$)

T_{WC}	Time Slot	P_1 T_{Sim}	P_2 T_{Sim}	P_3 T_{Sim}
0	1	0.01		
1			0.03	0.04
3			0.05	0.06
5	2			
6			1.03	1.04
8			1.05	1.06
10				

Figure 27. Multi-processor event sequence with optimal ΔT_{max}.

As shown in Figure 28, when ΔT_{max} is greater than 1, the first time slot's transmission process and sensing processes are allowed to run concurrently. Furthermore, the second time slot's transmission process is allowed to run directly after the first transmission process. While it appears that the validity of the second time slot may be intact, first time slot results are obviously compromised.

Three Processors ($1 < \Delta T_{Max}$)

T_{WC}	Time Slot	P_1 T_{Sim}	P_2 T_{Sim}	P_3 T_{Sim}
0	1	0.01	0.03	0.04
1	2			
2			0.05	0.06
4			1.03	1.04
6			1.05	1.06
8				

Figure 28. Multi-processor event sequence with large ΔT_{max}.

The objective when migrating a single processor simulation to a multi-computer environment is to achieve smaller run times while maintaining validity. ΔT_{max} is tuned on a per simulation basis in order to achieve these goals. As shown above, a ΔT_{max} of zero effectively forces the simulation to run sequentially with little or no opportunity for parallel processing. It would likely run slower on multiple processors than it would on a single processor. As ΔT_{max} is increased, more processing can take place in parallel before the simulations have to perform a time resynchronization, and thus efficiency increases. Hardware architecture efficiency curves E_{A1} and E_{A2} in Figure 29 show this trend. They also show how increased communication speeds vary these curves.

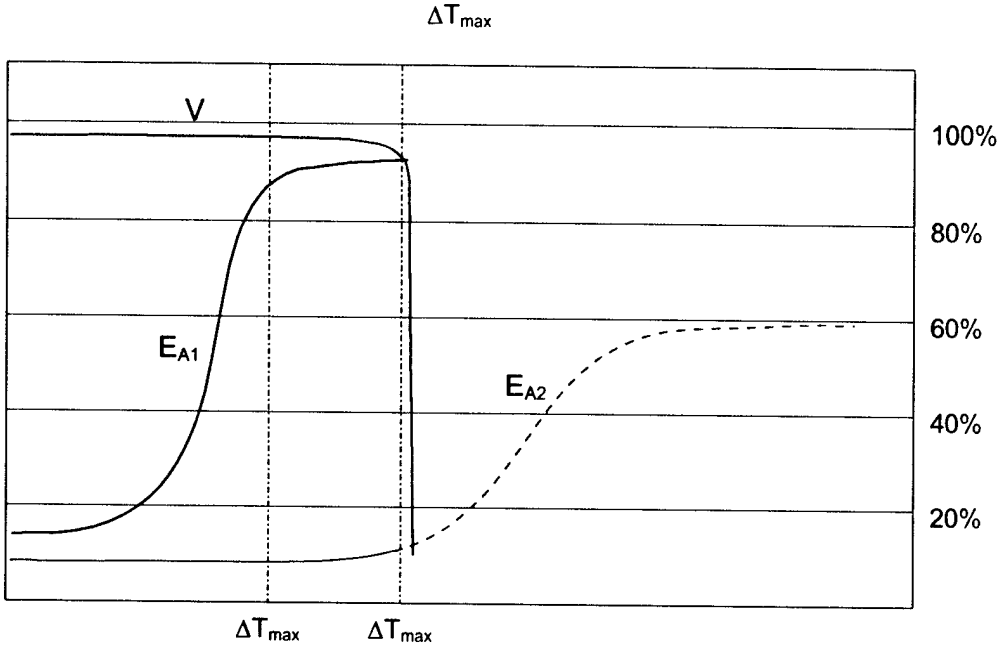


Figure 29. Theoretical ΔT_{max} vs validity and efficiency graph.

As ΔT_{max} is further increased, various activities on some processors are allowed to start before other activities, which must be completed first in order to insure validity, finish processing. The negative effect of this situation on validity varies differently in different simulations. In some simulations, the effect is gradual; in others, the effect is immediate, the simulation goes unstable, and no useful results can be derived. Furthermore, the more inherently parallel a simulation is, the more likely that ΔT_{max} can be increased to a point that delivers very high efficiency, while still remaining well below the point where results would become invalid. Validity curve V in Figure 29 shows how validity drops off quickly after some optimal ΔT_{max}.

10.5 ANALYSIS OF EXPERIMENTAL RESULTS

The chart of ΔT_{max} vs. efficiency and validity shown in Figure 30 is taken from the actual results of varying ΔT_{max} in multiple runs of the NFSS OMS multi-computer simulation across the twelve-processor cluster. The architecture 1 (A1) curve was generated with the network running in 1000 Mbps mode. The A2 curve was generated with the network running in 100 Mbps mode. As can be seen in the chart, efficiency increases rapidly from 25 to approximately 93 percent as ΔT_{max} is varied from .2 milliseconds to 200 milliseconds in architecture 1. In architecture 2, efficiency never gets above 25%. It is not until ΔT_{max} is increased to over 1 second, that validity begins to suffer.

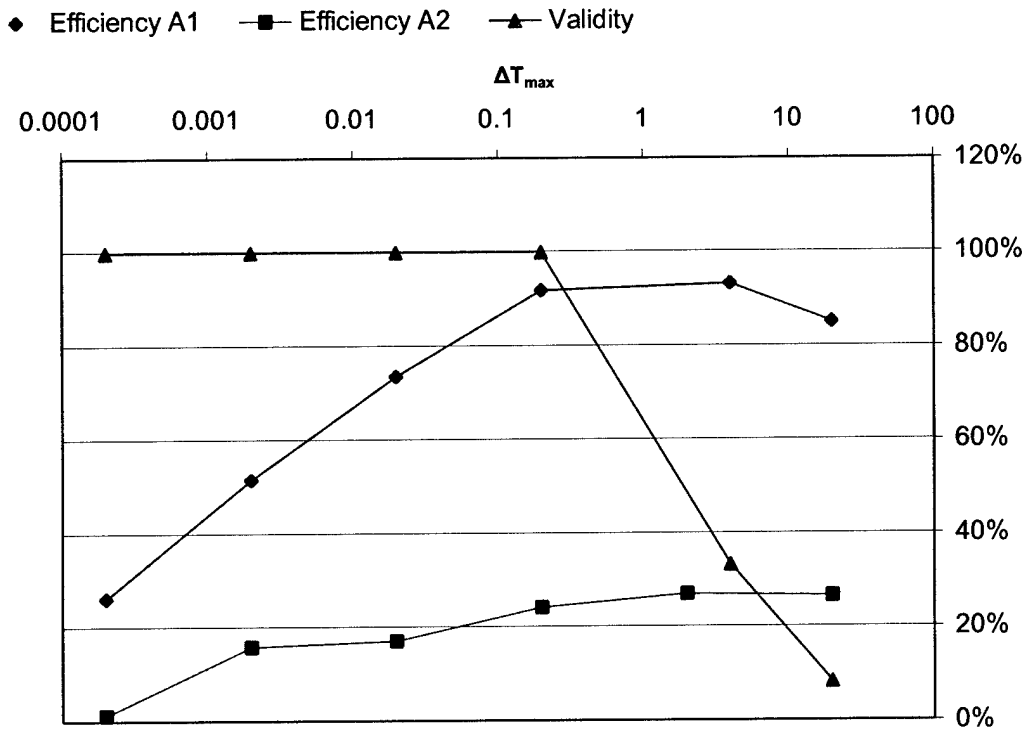


Figure 30. Experimental ΔT_{max} vs validity and efficiency graph.

11. SUMMARY OF EFFORTS

As part of this Phase 1 effort, PSI has completed all of the key objectives defined in its proposal. In particular, we have analyzed and documented our efforts in the following areas:

- Architectural Use Of Tasks, Subtasks, Modules And Threads - Analyzed the use of hierarchies of tasks, modules, and threads in both military and commercial applications, including large sensor fusion systems, planning tools requiring large embedded simulations with optimization, transaction processing systems, database processing systems, etc.

- Instanced Module Facility - Completed the detailed design of the automated instance module facility for the parallel processing version of VSE and GSS. Implemented the automated model instance facility in the current production version of VSE and GSS.

- Standards For Maximizing Module Independence - Designed the implementation of automatic enforcement of software architectural standards to ensure maximum independence of both instanced and non-instanced modules.

- Independent Module Database And Measures - Designed the independent module database, including built-in measures of module independence for both instanced and non-instanced modules.

- Maintaining Coherency Of Interprocessor Resources - Designed the run-time system for maintaining interprocessor resource coherency.

- Optimal Ordering/Scheduling Of Threads - Designed the run-time system for optimal ordering and scheduling of threads for both simulations and software tasks.

- Hardware-Software Architecture Compatibility - Analyzed pertinent MPP hardware architectures as they pertain to the design efforts on this project to ensure compatibility of software and hardware architectures.

The most significant findings of our Phase I effort are presented in the attached paper entitled: *The Effects of Parallel Processing Architectures on Discrete Event Simulation.*

11.1 USE OF THE NFSS OMS AS A TEST BED

In addition to completing the above objectives, PSI has provided a demonstration of the proposed technology using the Netted Full Spectrum Sensor (NFSS) Operational Management System (OMS). The NFSS OMS is being developed by PSI for the Intelligence and Information Warfare Directorate (I2WD) of the U.S. Army's Communications Electronics Command (CECOM) at Ft. Monmouth, NJ.

The NFSS-OMS allows engineering decision-makers to perform tradeoffs in sensor requirements and design parameters and to support staff officer decisions in real-time mission planning and execution. The graphical interface displays the current state of Blue Force (BLUFOR) and Opposing Force (OPFOR) as a continually updating Common Operational Picture (COP). The NFSS-OMS supports simultaneous connections from multiple systems providing sensor data. The accuracy and level of detail from each of the sensor systems vary with their capabilities.

As described in Section 10.3, the hierarchical sensor fusion process easily lends itself to the benefits of parallel processing. The hierarchical fusion stages have little dependence on each other. In addition, the lower layers can be separated into groups that have little or no dependence on each other. For example, each individual sensor system may have its data fused separately in the first level of data fusion and then fused with other sensor systems' data in subsequent levels of fusion. This demonstration used from 1 to 12 processing nodes and was instrumented to illustrate the potential efficiencies to be gained.

11.2 IMPLEMENTATION OF THE INSTANCED MODULE FACILITY

As part of Phase I, PSI has implemented the instanced module facility into its current production version of GSS and VSE. This facility removes the need for the software developer to deal explicitly with tables of attributes that relate to multiple instanced modules. For example, in a node and link simulation, each node can be represented by the same model. This facility allows the modeler to declare the number of instances at the module level. Then one can build a single instance without reference to the particular instance pointers of that module, except at the boundaries when one instance interacts with another module instance, in which case the specific instance is known.

This facility makes it easier to build multiple instanced modules since pointers can be eliminated. Some additional memory may be used, since software designers can reuse the same attributes for multiple instances. However, this makes the code more abstract, and more difficult to relate to the real system. These abstractions also remove the independence properties that afford parallelism. The amount of additional memory used is considered insignificant compared to the total. More importantly, the instanced module facility is considered a keystone to automating the software facilities required to produce efficient parallel processing architectures.

12. PHASE II PLANS

CATEGORIES OF SOLUTIONS

PSI has supported many parallel processing efforts using different technologies. These include clusters, DIS, and HLA. Figure 31 is an attempt to bring to light the many dimensions of the problem when considering parallel processing solutions to achieve significant speed improvements. An explanation of each column is provided below.

Configuration	Speed	Hardware Cost	Software Development Cost	Software Maintenance Cost
Single Processor	SP_S	SP_H	SP_D	SP_M
Multi-Computer Distributed	DI_S	DI_H	DI_D	DI_M
Multi-Computer MPP Cluster	CL_S	CL_H	CL_D	CL_M
Single-OS MPP	SO_S	SO_H	SO_D	SO_M

Figure 31. Categories of Solutions.

Configuration

The configuration categories are defined below. This is not meant to be an all-inclusive set - combinations could be used.

- **Single Processor** - This is used as a bench mark against which the other approaches can be measured.
- **Multi-Computer Distributed** - This covers the multiple computer case using distributed simulation, e.g., DIS and HLA. While PSI considers HLA an interoperability solution and not a parallel processing solution, it is addressed in Figure 31 because some people consider it a parallel computing approach. The main characteristics are separate operating systems and wide area (geographically spread) networks.
- **Multi-Computer MPP Cluster** - This covers multiple computer clusters which can range from a large Beowulf cluster to a more tightly coupled set of computer chips, each running their own operating system. The main characteristics are separate operating - systems and high speed LANs.
- **Single-OS MPP** - This implies that a large set of processors are controlled by a single operating system. PSI also considers this category as having local memory to each processor with fast hardware caching and coherency protection across processors.

Additional combinations exist and can be developed. However, as indicated above, this list is considered representative for the purposes of this discussion.

Speed

The single processor has been used to provide a benchmark on speed. It is convenient to use 1 GigaHertz (GHz) as the baseline, since most of today's single processor machines exceed this. Also, speed up using multiple processors depends upon the inherent parallelism of the system being modeled. We are concerned with partially independent systems. This implies they are not embarrassingly parallel.

As an example of partial independence, we are concerned with large military systems that interchange data somewhat randomly - on the order of thousands of messages per (simulated) second, with synchronization required on the order of every hundred messages. This implies a significant message exchange over a 4 hour scenario. This must be accomplished in approximately one minute of real time to perform the analyses required.

Another way to look at this is to consider that 50M to 100M messages must be transferred somewhat randomly among parallel processors in approximately one minute, with synchronization every 100 messages. We must also consider that the amount of time spent transferring the messages is about 100 times less than the other processing that must be performed within each processor to transmit a message. Therefore there is a significant degree of partial independence.

PSI's experimental results, as analyzed in the attached paper, Appendix A, indicate that *partially independent* systems will run several times faster on single-OS machines than on multi-OS machines. This is because of the latency encountered - going across the layers of protocol from one operating system to another - when cross-scheduling, sending messages, and synchronizing simulation clocks. For these types of systems, feasibility - driven by validity - is the key issue. Speed comparisons are provided below.

- **Single Processor** - Single processor running time is used as a bench mark. Cannot provide a feasible solution.
- **Multi-Computer Distributed** - Latency is huge. Cannot provide a feasible solution, independent of number of processors.
- **Multi-Computer MPP Cluster** - Measured in the attached paper. Latency plays a major role in meeting the combination of validity and speed requirements. We anticipate a problem achieving feasibility as the number of processors grows above 1000.
- **Single-OS MPP** - PSI considers this category as having the best speed performance by a potentially wide margin over the others, and therefore having the best chance for achieving a feasible solution. However, to achieve significant speed margins will require the kind of software solution described here for partially independent systems.

Hardware Cost

This refers to the costs of different hardware architectures to achieve a feasible solution. Hardware cost comparisons are provided below.

- **Single Processor** - Single processor can be used as a bench mark on cost.
- **Multi-Computer Distributed** - Number of processors multiplied by single processor cost.
- **Multi-Computer MPP Cluster** - Number of processors multiplied by single processor cost plus some overhead cost.
- **Single-OS MPP** - It is not unusual to spend several times more per processor on single-OS processors than multi-OS processors. However, the number of processors needed to achieve a feasible solution may be sufficiently less, potentially mitigating the cost differential or providing a favorable cost.

If a feasible solution is not achieved, there is no savings. Finally, one must consider the redundancy required to cope with down-time for each of these systems.

Software Development Costs

Software development cost must include consideration for time to get to the solution, and the considerable history of prior projects that have not achieved their goals. Conventional approaches to parallel processing software generally include many explicit parallel processing user code calls. Because of this manual tailoring, and the typical level of complexity encountered, the cost of developing simulations to take advantage of parallel processing platforms is significantly greater than the cost of developing single processor simulations. Software development cost comparisons are provided below.

- **Single Processor** - Single processor can be used as a bench mark on cost.
- **Multi-Computer Distributed** - Some overhead costs are incurred, but likely not to be more than 50% higher than the single processor cost when building the software. Using GSS can bring this cost down to the single processor cost. Time and cost for testing and debugging can run very high just due to the coordination required in the distributed environment.
- **Multi-Computer MPP Cluster** - Special approaches taken to tailor the solution to the number of processors to alleviate extremely high costs. When testing and debugging is considered, reduced costs still can be whole number multipliers above the single processor cost. Project failure rates are high. Using GSS, this cost can be brought down to the single processor cost.
- **Single-OS MPP** - Very special approaches taken to tailor the solution to the number of processors to alleviate extremely high costs. When testing and debugging is considered, reduced costs still can be whole number multipliers above the single processor cost. Project failure rates are very high. Using GSS, this cost can be brought down to the single processor cost.

The GSS solution leverages modeler knowledge in a way that allows user code to be independent of the intended runtime platform. Regardless of the number of processors, number of operating systems or operating system type, GSS user code will remain constant.

The complete GSS solution proposed for Phase II automates load management, both for distribution and balancing. Along with other the PSI software technologies, the development cost implications are profound, especially when confronted with finding feasible solutions for the types of partially independent problems that PSI has been working on.

Software Maintenance Costs

Software maintenance cost must include consideration for time to get to get back on the air. Software maintenance cost comparisons are provided below.

- **Single Processor** - Single processor can be used as a bench mark on cost.
- **Multi-Computer Distributed** - Some overhead costs are incurred, but likely not to be more than 50% higher than the single processor cost when making changes to the software. Using GSS can bring this cost down to the single processor cost. Time and cost for testing and debugging can run very high just due to the coordination required in the distributed environment.
- **Multi-Computer MPP Cluster** - Because of the special approaches taken to tailor the software to the number of processors, it becomes difficult to make changes to the functional logic due to the typical level of complexity encountered. When testing and debugging is considered, costs can be whole number multipliers above the single processor cost. Using GSS, this cost can be brought down to the single processor cost.
- **Single-OS MPP** - Because of the special parallel processing software abstractions that become part of the user code, it is difficult to make changes to what otherwise may be simple functional logic. The typical level of complexity encountered is very high, especially when testing and debugging is considered. Costs can be whole number multipliers above the single processor cost. Using GSS, this cost can be brought down to the single processor cost.

SUPPORT FOR ALL SOLUTION APPROACHES

There is a very large class of problems that have a high degree of parallelism but are far from being embarrassingly parallel. We have labeled these problems as *partially independent.* PSI has been working on a subset of this class that requires huge multipliers on speed (on the order of thousands) over that of a single processor to allow simulations to be run at speeds that exceed real time by factors of 10 to 100 and higher.

The distributed multi-computer approach used for DIS and HLA simulations have been designed to bring together separately developed simulations running in real time. Although they are not suited to solving this type of problem, GSS currently supports these approaches.

Multi-computer MPP clusters can be implemented in various ways. We have compared the case where each processor has its own operating system to the MPP single OS case. Combinations of single OS and multi-OS machines can be analyzed as being in between these two cases. In our experiments, we have determined that the latency encountered to move data across the OS boundaries becomes significant when simulating partially independent systems. Even though GSS provides great software simplifications for either type of parallel processing approach, feasible solutions that meet the speed and validity constraints will likely not be achieved for a class of problems in the near future without the single OS solution.

Based upon the above analysis, and the results derived in the attached paper, a single-OS MPP is best suited to meet speed and validity requirements for simulations of large partially independent systems that must run much faster than real time. The high software development and maintenance costs, and the inability to take advantage of extremely small latencies provided by this architecture have heretofore made it hard to justify. These problems can be overcome by GSS, making the single-OS MPP most attractive for simulating partially independent systems.

Most importantly, GSS solutions provide significant increases in productivity for all of the above types of processor architectures. The instanced model facilities can be taken advantage of on a single processor as well as all of the parallel processor approaches.

GSS PARALLEL PROCESSING DEVELOPMENT ROADMAP

In 1996, PSI was awarded a contract from CECOM's Software Engineering Center (SEC) for Visual Software Development For Parallel Machines, [13]. This contract along with a subsequent DARPA MHPCC BAA Consortium contract, [14], supported the development of the *GSS* Multi-OS environment illustrated in Figure 32. We now describe the work to be completed.

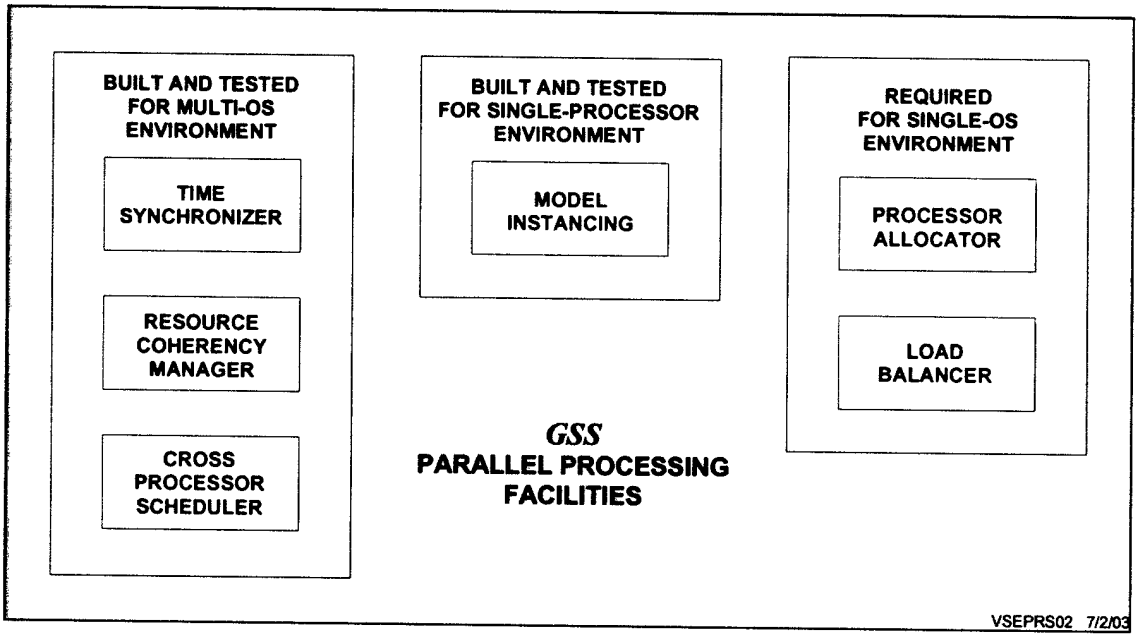


Figure 32. Complete GSS parallel processing solution.

Work To Be Completed In Phase II

In Phase II, we will build and test candidate design approaches using actual systems and simulations, leading to a selected product for Phase III. Having already performed an analysis of this problem in Phase I, we have determined that the use of an MPP running a Single-OS provides the best environment to complete the product line. This selection is based upon the current need for simulations of partially independent systems that run much faster than real-time. Building the intelligent adaptive software required to take advantage of the low latencies of this architecture is key to ease-of-use as well as the practical application of this architecture for a wide range of problems. Finally, the solutions for automatic processor allocation and load balancing will also support a multi-OS architecture, although the inherent latencies of these systems will not be nearly as efficient for a wide range of problems.

Completion of each of the elements of the solution in Phase II is described below.

- **Time Synchronizer** - The time synchronizer has been built on prior projects, and has been tested in Phase I as well. Porting this facility to a single-OS machine should not be a significant effort.
- **Resource Coherency Manager** - The resource coherency manager has been built on prior projects, and has been tested in Phase I as well. Porting this facility to a single-OS machine should not be a significant effort. If OS facilities exist for fast caching data between processors, these may be invoked and will have to be tested.
- **Cross-Processor Scheduler** - The cross-processor scheduler has been built on prior projects, and has been tested in Phase I as well. Porting this facility to a single-OS machine should not be a significant effort.
- **Model Instancing** - The model instancing facility has been built and tested in Phase I. Porting this facility to a single-OS machine should not be a significant effort. Changes to the run-time database will have to be made and tested.
- **Processor Allocator** - The processor allocator has been designed in prior projects. It must be built and tested in Phase II.
- **Load Balancing** - The load balancing facility has been designed in prior projects. It must be built and tested in Phase II.
- **Run-Time Initialization** -

In addition to the above pieces, the run-time initialization facilities will have to be modified for a single-OS parallel processing environment. This must provide for initialization of databases and initiation of functions to support the above items.

Commercialization

PSI has elected to work with SGI in Phase II. This will provide for a machine at PSI's facilities in Spring Lake, NJ. SGI will also provide systems engineering assistance, particularly in the area of operating system calls and optimization of resource allocation.

PSI plans to use large existing battlefield simulations that will be tied together as a single simulation to provide for a real test bed environment. These simulations will demonstrate the use of parallel processing to support important future applications for military planning. It will also demonstrate an approach that can be extended to the industrial and commercial business planning process.

Finally, SGI will initiate initial marketing efforts of the parallel processing software product to existing and prospective clients.

Benefits Of The Final Product

The major benefits of the product to be produced in Phase II are illustrated in Figure 33. The top bar contains the three constraints that must be satisfied before parallel processing can achieve broad commercial success. With a software environment that is easy to use, modelers and software developers trying to obtain a combination of high speed and efficiency will be able start to focus on valid solutions to the real problems they are trying to solve. If their software can run on a wide variety of platforms without modification, they will have more choices and more competition will ensue. The paradigm defined in Figure 33 is key to ensuring speed, validity, and ease-of-use

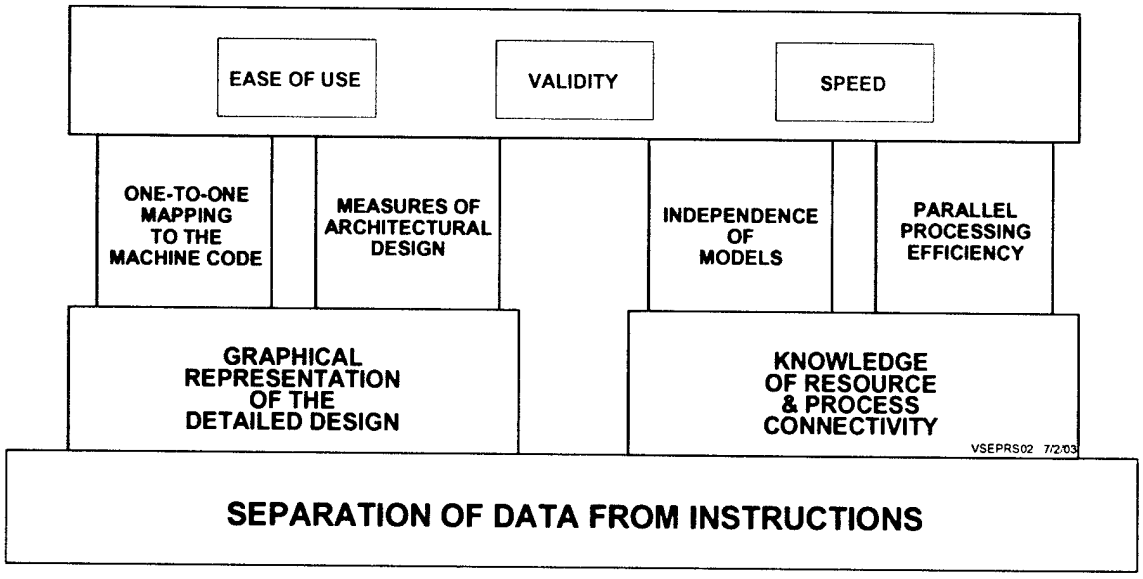


Figure 33. Illustration of how the basic paradigm supports the objectives.

13. REFERENCES

[1] Patterson, D., Bell, G., et al, "Massively Parallel Uproar," Upside, pps 88-97, March, 1992.

[2] Bell, C. G., "Ultracomputers A Teraflop Before Its Time," Communications of the ACM, August 1992, Vol.35, No 8.

[3] Mitchell, R., "In Supercomputing, Superconfusion," Business Week, pps 89-90, March, 1993.

[4] *Netted Full Spectrum Sensor (NFSS) Operational Management System (OMS)*, Final Report, Contract DAAB07-02-C-l415, U.S. Army CECOM/I2W , 30 July 2002.

[5] GSS User Reference Manual, Prediction Systems, Inc., Spring Lake, NJ, 2003.

[6] Athans, M. and Falb, P.L., *Optimal Control*, McGraw-Hill, New York, 1966.

[7] Gelb, A., Editor, *Applied Optimal Estimation*, MIT Press, Cambridge, MA, 1974.

[8] Gordon, G., *A General Purpose Systems Simulation Program*, Proc. EJCC, Washington, D.C., pp 87-104., MacMillan Publishing Co., New York, 1961.

[9] Gordon, G., **The Application of GPSS V to Discrete System Simulation**, Prentice Hall, Englewood Cliffs, NJ, 1961.

[10] Cave, W.C., *Simulation of Complex Systems*, Prediction Systems, Inc., Spring Lake, NJ, June 2001.

[11] Ramadge, P.J. and W.M. Wonham, "Supervisory Control of a class of discrete-event processes," SIAM J. Control Optimization, vol 25, no.1, pp 206-230, Jan. 1987.

[12] Ramadge, P.J. and W.M. Wonham, "The Control Of Discrete-Event Systems," Proc. IEEE, vol 77, no.1, pp 206-230, Jan. 1989.

[13] *Visual Software Development For Parallel Machines*, Final Report, Contract DAAB07-97-C-H501, Software Engineering Center, U.S. Army CECOM, March 31, 1997.

[14] *Multi-Computer Version of GSS*, Final Report to University of New Mexico to support the DARPA BAA MHPCC Consortium, September 31, 1998.

APPENDIX A

THE EFFECTS OF PARALLEL PROCESSING ARCHITECTURES ON DISCRETE EVENT SIMULATION

W. Cave, E. Slatt, & R. Wassmer

June 19, 2003

PREDICTION SYSTEMS, INC.
PREDICTION & CONTROL SYSTEMS ENGINEERS

309 Morris Avenue
Spring Lake, NJ 07762

(732)449-6800 (732)449-0897
psi@predictsys.com www.predictsys.com

OVERVIEW

We are concerned with simulation as it is used to support design decisions, evaluate system effectiveness and performance, and predict future outcomes resulting from different actions. We limit this concern to simulations that must produce valid results in specified time frames to be useful. This implies that modelers must provide a level of detail that achieves sufficient model accuracy to produce valid simulation results.

As systems become more complex, particularly those containing embedded decision algorithms, mathematical modeling presents a rigid framework that often impedes representation to a sufficient level of detail. Using discrete event simulation, one can build models that more closely represent physical reality, with actual algorithms incorporated in the simulations. Higher levels of detail increase simulation run time. This paper is focused on the effects of model architecture, run-time software architecture, and parallel processor architecture on speed.

Simulation Run-Time Constraints

Parallel processing is used to meet time constraints that cannot be met running on a single processor. Time constraints occur for many reasons. If it takes many days to complete one simulation run, then it may take weeks of running simulations to obtain a single valid test. This can inhibit the simulation development and validation process. If one wants to use simulated output to make a decision within hours, having to wait days renders the output useless.

Cases of interest involve modeling real world systems where the time to run a single processor simulation far exceeds the time taken by the real system for the same scenario. One must question why a simulation cannot run at least as fast - or even much faster, using a large number of parallel processors. Such applications cover many fields, including computer design, communication network design, and planning and control system design. When running applications requiring many hours of simulated scenario time, it is desirable to have the simulation time to real time ratio be very high. One can then run a simulation, review the output, make changes, and run another simulation quickly.

ENSURING VALIDITY

Test results can be presented in many ways, e.g., graphically observing events unfold, or visually scanning reports of data points. These can be classified as measures of merit, i.e., measures of effectiveness or performance of a system. We use M to denote a generalized vector of values measuring the properties of a system under test (SUT). Field or laboratory testing is subject to the Uncertainty Principle, and properties being tested are typically presented in terms of a distribution of measurements. At the end of a series of tests, M is represented as a set of distributions, one for each property or element, Vi, of the measurement vector. Typically, these distributions can be characterized in terms of a mean and variance as illustrated in Figure 1. These distributions are used to determine validity of test results.

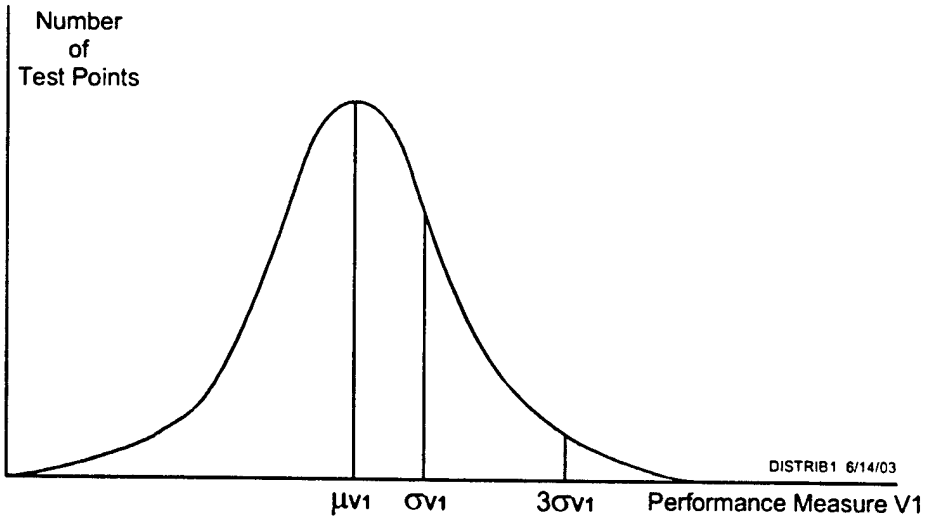


Figure 1. Illustration of the distribution of test results for performance measure V1.

When field or laboratory testing is prohibitive or expensive, one typically resorts to simulation. One must then assess the cost of obtaining valid results from a simulation.

Comparing Statistical Results

When running a single processor simulation, one will get the same results from every run unless it is taking in real time data. To provide a more accurate view of results, values of parameters that may vary are drawn randomly from predetermined distributions representing known or anticipated variations. To analyze the effects of these variations, one typically runs Monte Carlo simulations, whereby the simulation is run a sufficient number of times, each with a different random number seed, to produce a distribution of results. Then one can compare the distribution of the measurement vector from the single processor simulation, Ms, to that of a valid test set, Mt. If the distributions are deemed to be the same by the "validation committee," then the simulation can be used as a valid substitute for field or laboratory tests.

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

Validating Parallel Processor Simulations

As indicated above, we will start with a validated single processor simulation and investigate the potential loss of validity when running that simulation on a parallel processor. The major factor of concern is *data coherency*. Data coherency implies that, when a *process* (set of instructions) accesses data in memory, the data has *not* been changed by a previous process in a way that causes the logic to produce intermediate results that lead to invalid simulation results.

The data coherency problem is not limited to discrete event simulation. It can occur in software. For example, if a process on one processor shares a data structure with a process on another processor, and they both update that data structure thinking the values are unchanged the next time they access that data structure, the results of either of these processes may be invalid. Clearly the end results depend upon the logic in the processes, and the assumptions that logic makes regarding the coherency of the data. In software, this problem is generally solved by ensuring that, while a process is running, no other process shares its data - unless by design.

Simulation Time Synchronization

In discrete event simulation, processes are scheduled at specified (event) times in the future, with the anticipation that the data accessed by these processes will be correct at those scheduled times. Loss of data coherency can occur due to loss of *time synchronization*, where time is simulation clock time. If a single processor version of a simulation is producing valid results, with no data coherency problems, then it can run on a parallel processor and produce the same results, provided the following is true: Processes running on different processors and sharing data run in the same sequence as they would on a single processor. This is a sufficient - but not a necessary condition for validity of the results.

To validate the results of a simulation, one must compare the distributions of the measurement vector from the parallel processor simulation, Mp, to that of a valid test set, Mt, or valid single processor simulation, Ms (single thread implied). Validity can be achieved without having the same process sequence. In fact, the process sequence will vary in the single processor case simply by varying the random number seed. As stated above, maintaining a strict sequence is not a necessary condition. It is important to know what will produce valid results from a simulation, and what happens when we move that simulation from a single processing environment to a parallel processing environment.

INHERENT PARALLELISM IN SYSTEMS

In many applications, the best solution approaches do not lend themselves to parallel processing. For example, the fastest known algorithms for sparse matrix inversion are inherently sequential. These methods, known as symbolic preprocessors, eliminate looping and testing, leaving only the minimum sequential set of add, subtract, multiply, and divide operations to be performed, see for example, Berry, [1], and Hachtel, [2]. In addition, focusing on parallelism in short instruction strings inside program loops does not necessarily lead to efficient use of large numbers of processors. The overhead required to control which processor will perform what set of instructions using what data may take as long as the user instruction strings themselves, see Reiher [3].

Increases in processing speed clearly depend upon the inherent parallelism of a system as well as the solution approach. If the problem has little inherent parallelism, i.e., each step must follow in sequence with each depending upon the prior outcome, then parallel processors will not help speed up the solution. If large blocks of code can be processed at the same time, independently, parallel processor computers may significantly improve the speed. To realize such speed increases, one must take effective advantage of the inherent parallelism of the system when designing model and simulation architectures.

Properties Of Independence

We will focus on discrete event simulation. To this end, we define the property of process *independence* as follows: Processes are independent if they can be run concurrently without loss of validity.

An example of independence occurs when performing Monte Carlo analysis. If a simulation is run N times, such that each run (n) uses a different random number seed and produces a final output containing the measurement vector Mn, then each simulation can be run concurrently producing the same set of measurement vectors M_1, M_2, ..., M_N as would be produced if run sequentially. In this example, outputs are compared after the runs are complete, but there is no data shared during the course of the simulation. The results are the same because each simulation is independent of the others. Hence, every process in one simulation is independent of every process in any of the others. This is known as an "embarrassingly parallel" example. It is only of interest here as an extreme case of total independence.

Partial Independence

The discrete event simulations of interest are of systems with partially independent components. Examples include networks of physical systems, typically connected by communication equipment. Large networks involve hundreds or even thousands of complex components, each running concurrently. An example is air traffic control. Aircraft platforms may require 6 degree of freedom models. These platforms interact via radar sensors and wireless communication systems. Signals and messages are continually being interchanged by these models during the course of a simulation, just as they do in a real system.

In such simulations, there is a large amount of data sharing, much of which is highly synchronized, causing major concerns about validity - even in a single processor simulation. Yet, there is generally a large degree of inherent parallelism. This is supported by the fact that single processor simulation scenarios typically take considerably longer than their real system (parallelized) counter parts. It is not unusual for a simulated two hour scenario to take days.

Building models that take advantage of the partial independence of systems has heretofore been a challenge. This is a very large application area where PSI applies most of its efforts. To address the problem of ensuring validity, as well as modeling partially independent systems on parallel processors, PSI developed the General Simulation System (GSS), [4].

THE GSS - PARALLEL PROCESSING ENVIRONMENT

GSS has been designed to support efficient parallel processing, both during development and run-time. There are a number of system features that support these design goals.

1. Data structures (resources) are separated from instructions (processes) to track which processes share what resources. This is used to determine independence and thus the potential for concurrent processing at run-time.
2. Models are built using a visual CAD tool, where icons of processes and resources are connected by lines, denoting data sharing. This provides a one-to-one mapping from engineering drawings to the code. Double clicking on an icon brings up the code.
3. Model connectivity (independence) can be inspected visually to determine if the inherent parallelism of a physical system is properly represented. Thus, models do not have to be changed to move from a single processor to a parallel processor. No code is changed - a major factor affecting validity.
4. Knowledge of the architectural parallelism is stored within the system, and used during run time. To take advantage of this knowledge, the system has its own run-time environment that allocates processors to processes in a maximally efficient way.
5. Efficient data coherency protocols ensure that processes are not using the same *resources* (data structures) at the same time. These can be tailored to manage hardware coherency protocols on a parallel processor.
6. The scope of a resource is very large compared to the way attributes are formulated in typical programming languages, e.g., C, C^{++}, Java, FORTRAN, etc. Similarly, processes have a large scope compared to subroutines in other languages. This is due to the additional level of hierarchy in both. This provides substantially increased scalability, and much larger bound instruction sets.
7. The run-time system provides for efficient cross-scheduling of processes across processors as well as fast scheduling within a processor.
8. The simulation clock on each processor does not vary by more than ΔT, a parameter specified by the modeler. Simulation clock units can vary from picoseconds to days in a single simulation. This allows the modeler to set ΔT to the maximum value that ensures validity of results. As described below, this is a key factor in obtaining efficiency of parallel processing, reducing idle time of processors waiting for clock synchronization.
9. The system supports instanced models. At run-time, large numbers of independent model instances are allocated to separate processors, with minimized resource sharing across processors.
10. The run-time system can perform optimized load balancing given real time data on processor loading.

Scheduling Processes

Modelers use a GSS SCHEDULE statement to cause processes to be placed in a schedule queue to be run at a specified time in the future. Processes scheduled at the same time can be given a priority. If no priority is assigned, it is implied that the order does not matter, i.e., a valid result will occur if ordered randomly.

Valid results can occur even when processes, scheduled at different times, are run out of order. This is because timing may or may not affect validity. For example, if message A comes in before message B, but neither get processed until both are in, it does not matter which one comes in first. Or, if ten messages must come in before something happens, which ones get in under the wire may not matter because, in the real world, the results are valid either way. This is especially true when variations occur naturally, causing the distribution of the measurement vector.

Determining The Maximum Value Of ΔT

As indicated in feature 8 on the prior page, one must determine the maximum value of ΔT that still ensures validity of results. This is done by running multiple Monte Carlo sets of the simulation in the GSS parallel processing environment, varying ΔT until the measurement vector, Mp, produces distributions whose variance exceed those of the single processor version, Ms. Figure 2 illustrates this effect for two different simulations, A and B.

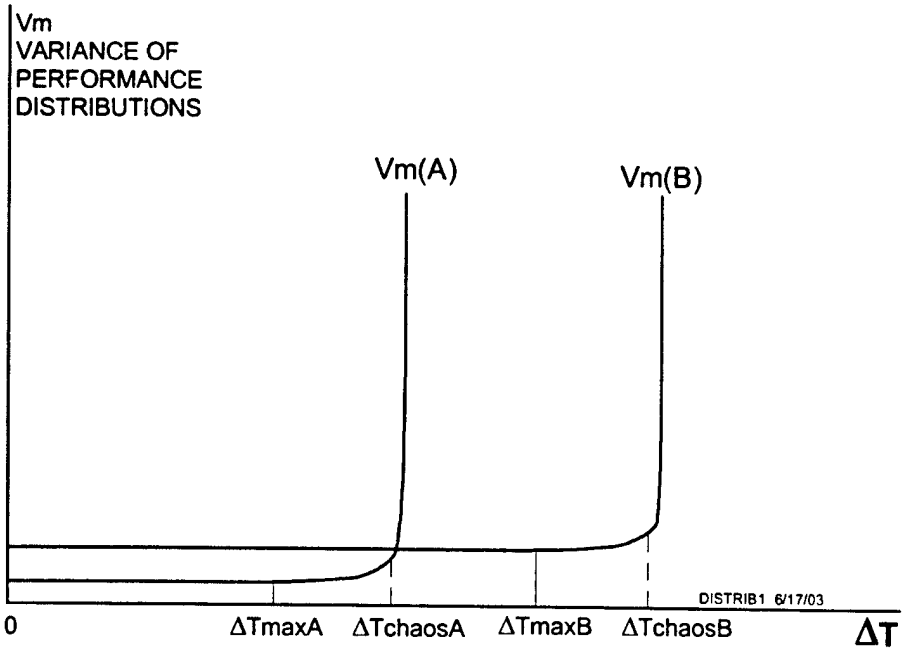


Figure 2. Illustration of the selection of ΔTmax.

For simulations of nonlinear systems, the variance of the distributions is typically unchanged until a break point, ΔTchaos, at which point results become chaotic. If the systems being simulated have a synchronized component, i.e., events occur on a time synchronized basis, then this effect can be expected to occur as ΔT crosses the time synchronization point. Judgment can be used to back off to a valid point. In the sensor simulations used to test this approach, [5], the break point occurred at about 1 second. Backing off to ΔTmax = 0.8 seconds was sufficient. It is likely that changes could be made to this simulation to move the curve to the right, increasing the allowed ΔTmax. If changes are made to a simulation, ΔTmax must be revalidated.

Simulations of TDMA wireless systems may place a more stringent requirement on ΔTmax. For example, military Link 16 networks use a JTIDS terminal with time slot synchronization at 7.8125 milliseconds. If the modeler has properly synchronized scheduled events within a time slot, a ΔTmax of 7 milliseconds is likely to ensure validity of simulations that model message traffic to the time slot level. Depending upon other items in the simulation, modeling to the JTIDS frame level may relax this requirement to more than 10 seconds, since a frame covers 12 seconds. But such a modeling approach would severely limit the validity of the simulation to investigate network performance when subject to rapid response requirements at the individual message level.

We note that validity as a function of ΔTmax is generally independent of the parallel processing environment. However, the actual curve will encounter variations resulting from the effects of random ordering of processes, producing a distribution of results caused by these random variations. This distribution should be within the simulation validity requirements.

The Effect Of ΔT On Parallel Processing Efficiency

Parallel processing efficiency is defined here as: the ratio of the time it takes to run a simulation on a single processor to that on a parallel processor, divided by the number of processors. As shown in Figure 3, ΔT plays a major role in processing efficiency.

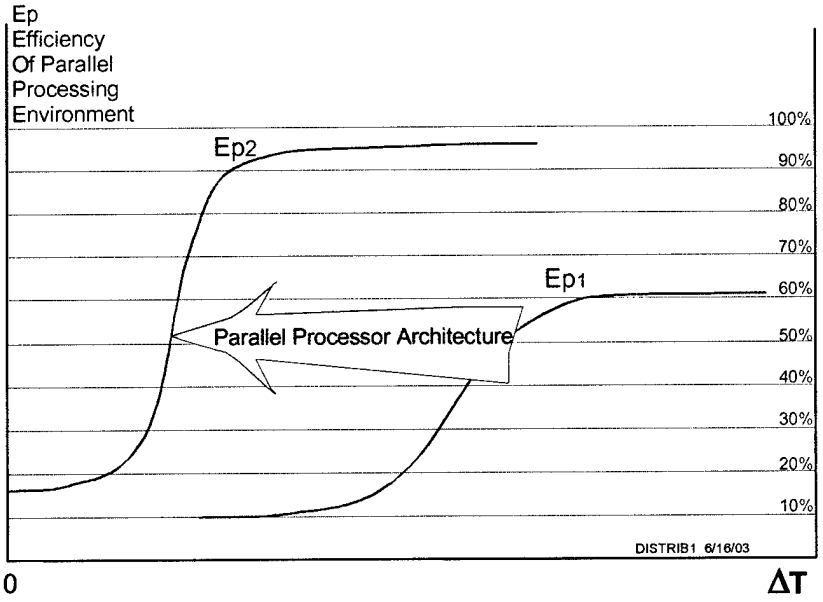


Figure 3. Illustration of parallel processor efficiency versus ΔTmax.

The Effects Of Architecture On Parallel Processing Efficiency

The curves shown in Figure 3 are representative of those derived from data taken on percent efficiency versus ΔT as part of the experiments reported upon in [5]. As ΔT is increased from 0 to larger values, efficiency increases as shown in each of the two curves, Ep1 and Ep2. These curves represent different parallel processor architectures for the same simulation and same number of processors. The difference in levels achieved represents the different losses of efficiency incurred when:

- Maintaining data coherency across processors
- Transferring shared data from one processor to another
- Scheduling processes on different processors
- Idling due to imbalanced loading

The first three contribute latencies that reduce efficiency as they become a greater percentage of the overall processing time. Imbalanced loading will be treated separately below.

The Effects Of Software Architecture

The curves in Figure 3 also depend upon the inherent parallelism in the system being modeled. In the Monte Carlo (embarrassingly parallel) case, the overhead of the first three bullets is generally unnecessary. If the processing time for each simulation were identical, the total processing time should be that of a single processor divided by the number of processors. If run on a set of separate computers, the efficiency would be 100%. If the processing times were not equal, then the total time would be equal to that of the one simulation with the longest running time; idle time would occur at the end of the others.

In the cases of interest, where there is only partial independence, there may be a large amount of scheduling and data sharing among processors relative to the embarrassingly parallel case. However, as in the real systems represented, this may still be small compared to the processing required within a processor. In these cases, two areas become important.

- Modeling and simulation architecture
- Parallel processor run-time management software

If the model architecture does not match the inherent parallelism of the actual system from an independence standpoint, efficiency will be lost relative to that of the actual system. This is particularly obvious when building software abstractions that cut across multiple hardware instances of a subsystem, causing bottlenecks in a parallel processor environment. Software abstractions are commonly used to save memory - a major pitfall in parallel processing. GSS provides a visualization of the architecture that can be used to eliminate this problem.

If the software architecture is mapped along physical lines but the run-time management software has no knowledge of this, then good model architectures will likely be scattered randomly across processors, losing the efficiency they could otherwise provide. The GSS run-time system is designed to take full advantage of parallelism within model architectures. It runs on clusters; however, our analysis demonstrates that it runs most efficiently on tightly coupled processors under a single Operating System (OS).

The Effect Of Hardware Architectures

Given that the model and run-time software architectural approach takes full advantage of the inherent parallelism of a system, we must ensure that if we select a hardware architecture, it will meet the time and validity constraints. To illustrate the selection process, we will use a simple example of run-time constraints whereby a 2 hour scenario must be run in less than 6 minutes to achieve the desired goal. This implies a simulation time to real time ratio of 20. If the simulation runs 3 times slower than real time on a single processor, it must run with greater than 60% efficiency using 100 processors to achieve the goal. This provides a speed up factor of 60 (one hour of single processor time is done in one minute).

Figure 4 illustrates two different hardware architectures, Ep1 and Ep2, as candidates to achieve the goal. In the case of Ep1, the efficiency remains at about 10% as we approach ΔTmax. This illustrates the effect of latency on achieving a feasible solution, i.e., meeting the time and validity constraints. Ep2 achieves in excess of 90% efficiency prior to reaching ΔTmax. For this example, Ep2 clearly achieves the goal.

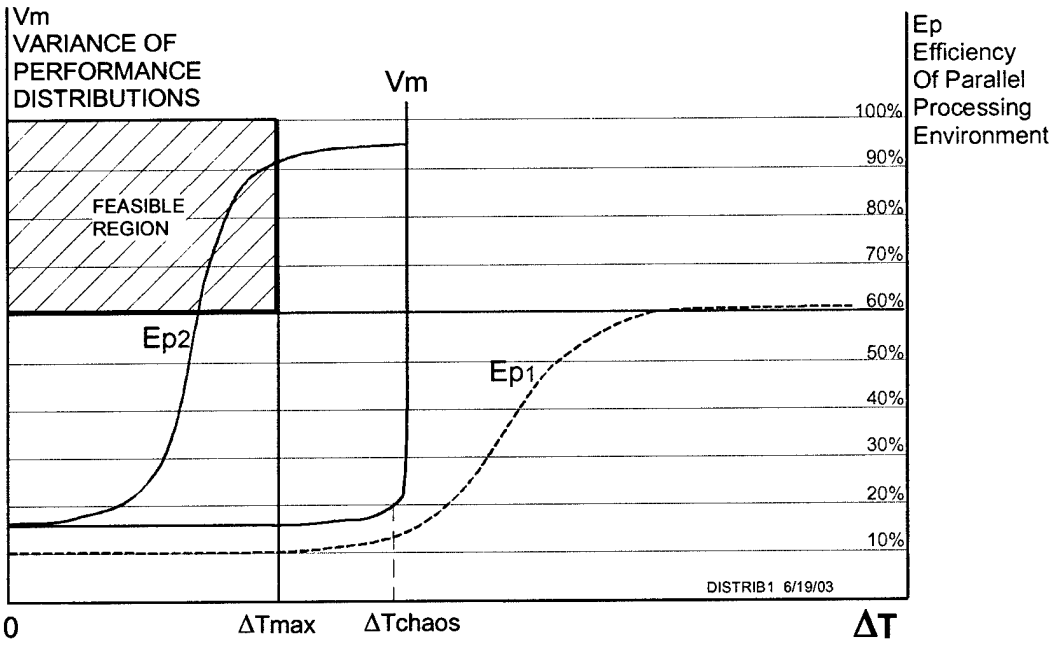


Figure 4. Relative effects of parallel processor architecture on validity.

When assessing parallel processing architectures, one must understand the dynamics of latency as it affects the feasibility of different hardware solutions. When simulating partially independent systems on parallel processors, one is prone to doing battle with chaos. These limits are apparent when using HLA federations to simulate large numbers of instances of complex units. When the federates are required to be highly synchronized to ensure validity, simulation times can quickly become excessive. There have been examples of parallel processing operating systems where substantial time is spent in special algorithms fighting chaotic behavior. These examples are prevalent in high latency systems.

LOAD BALANCING

When imbalanced loading occurs, decisions may be made regarding the movement of processes from processors experiencing maximum loading to processors experiencing minimum loading. If this is done dynamically, movement may waste more time than it can save. Consider the processor loading histogram in Figure 5. It has been ordered by loading level.

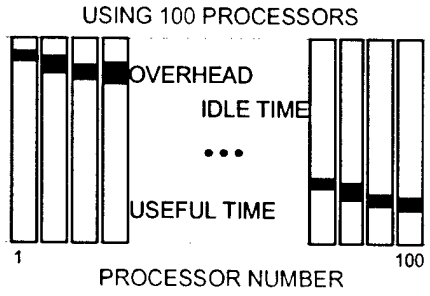


Figure 5. Ordered processor loading histogram.

If this histogram represents results of the total scenario, then it appears that many processors were idle a majority of the time. One may quickly conclude that many less processors could have been used to achieve the same run-time speed. This assumes that movement does not increase data sharing across processors, and that process loading remains in a steady state. If these assumptions are true, one may be able to hand tailor the placement of processes on processors to maximize processor utilization and therefore efficiency. This may be at the expense of speed loss that will be small if the assumptions hold. We consider this a rare case, and even then consider hand tailoring a questionable investment.

Now consider that Figure 5 represents a sample of a time period that is relatively short compared to the scenario. Also, consider that processor utilization changes dynamically during the scenario as shown in Figure 6. Then one must be concerned with the overhead of migrating processes verses the time constants of significant load changes, e.g., from times T1 to T2 to T3 in Figure 6. Given that the migration times are sufficiently small compared to the dynamic loading time constants, one still must be concerned about increases in data sharing across processors.

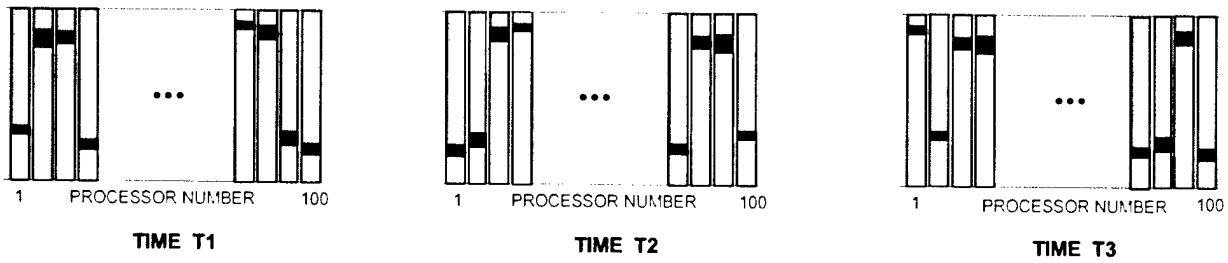


Figure 6. Changes in processor loading over time.

Taking Advantage Of The Independence Properties Of Actual Systems

By virtue of the physical architecture of engineered systems, components are designed to be maximally independent. This provides for system survivability as well as ease of maintenance. Therefore, models that are designed along physical lines, without software abstractions, can take advantage of this inherent independence. Knowledge of the partial independence (inherent parallelism) of models and their instances within a GSS simulation is used to support intelligent load balancing decisions, independent of the number of processors assigned.

Assigning And Migrating Models And Instances.

The visualization of models of components of physical systems in GSS provides the modeler with the ability to take maximum advantage of the partial independence of the system being modeled. Large models containing many layers of submodels can be instanced just as their physical counterparts. The automatic instancing facility built into the system ensures their independence, except at specified boundaries where instances share resources to represent the real world exchanges of partially independent entities.

All processes and resources within an instance are automatically assigned to the same processor by the run-time system. This ensures minimal data sharing across processors. Load balancing is achieved by comparing the dynamic load created by each instance to processor load dynamics. Thus, the time constants of each can be compared as well as the loading over selected time periods to determine if migration of an instance is likely to improve speed.

If latencies between processors in a large array vary sufficiently, then a latency matrix can be used to optimize the placement of instances. Assignment can be based upon the dynamics of data sharing between model instances.

Assignment and migration of model instances can be more efficient and more easily controlled when running under a single OS. When running in a cluster environment, GSS provides the same automatic coherency and cross-scheduling protocols between separate simulations interacting via a high speed network. It can also support multiple federates in an HLA environment, any of which may be GSS clusters or single OS parallel processors simultaneously.

Synchronization, Coherency, And Look-Ahead

When simulating large synchronized systems, it is not unusual for a large number (thousands) of processes to be scheduled at the same time and priority. When a process is popped off the schedule queue, it may have to wait for the use of a resource due to a coherency protocol. If the next process in the queue is scheduled at the same time and priority, and none of its resources are stopped by coherency protocol, it may proceed before the first. Additionally, algorithms can be added to the scheduler to determine the optimal ordering of processes within each processor that are at the same time and priority, e.g., optimal placement of those processes with interface resources. This form of look-ahead does not require retracing steps in an attempt to regain validity, but supports continuous forward motion in time while maintaining validity.

CONCLUSIONS

The potential speed increases that discrete event simulations can achieve on parallel processors depend directly upon:

- Model architecture
- Run-time software architecture
- Parallel processor hardware architecture.

The effectiveness of each of the above items depends upon the preceding one. Assessment of good hardware architectures can be masked by poor run-time software. Similarly, a poor model architecture will not take advantage of an excellent combination of run-time software running on a low latency parallel processor.

Many approaches to discrete event simulation using parallel processors disregard the hard constraints on validity, and the very real effects this has on efficiency. This is apparent from the literature, as many organizations continue to search for methods to unravel chaos instead of working to take full advantage of inherent parallelism and reduce latency.

Without an environment that affords modelers the ability to take maximum advantage of the inherent parallelism in partially independent systems, the information required to make a feasible (let alone optimal) selection of a hardware architecture may be masked.

When modelers find it easier to run simulations on a parallel processor than using their current approach on a single processor, the benefits of parallel processing will be finally realized. This implies that the environment they use must automatically take advantage of the inherent parallelism in partially independent systems.

REFERENCES

[1] Berry, R., "An Optimal Ordering of Electronic Equations for a Sparse Matrix Solution," IEEE Trans. on Circuit Theory, Vol CT-18, No.1, pps 40-50, January 1971.

[2] Hachtel, G. et al, "The Sparse Tableau Approach to Network Analysis and Design," IEEE Trans. on Circuit Theory, Vol.CT-18, No-1, January 1971.

[3] Rieher, P., "Parallel Simulation Using the Time Warp Operating System," Proceedings of the 1990 Winter Simulation Conference, New Orleans, LS, pp 38-45.

[4] GSS User Reference Manual, Prediction Systems, Inc., Spring Lake, NJ, 1996.

[5] *High Efficiency, Scalable Parallel Processing,* Phase I Final Report, DARPA Contract SB022-035, Prediction Systems, Inc., Spring Lake, NJ, June 2003.