The Effects of Parallel Processing Architectures on Discrete Event Simulation

William Cave[*a], Edward Slatt[b], Robert E. Wassmer[b]
[a]Prediction Systems, Inc. (PSI), Suite J, 309 Morris Avenue, Spring Lake, NJ 07762
[b]Prediction Systems, Inc. (PSI), Building 2, Suite 222, 3350 Highway 138, Wall, NJ 07719

ABSTRACT

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 also increase simulation run time. Hardware designers have succeeded in producing parallel and distributed processor computers with theoretical speeds well into the teraflop range. However, the practical use of these machines 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 paper addresses the application of parallel processing to simulations of real world systems of varying inherent parallelism. It provides a brief background in modeling and simulation validity and describes a parameter that can be used in discrete event simulation to vary opportunities for parallel processing at the expense of absolute time synchronization and is constrained by validity. It focuses on the effects of model architecture, run-time software architecture, and parallel processor architecture on speed, while providing an environment where modelers can achieve sufficient model accuracy to produce valid simulation results. It describes an approach to simulation development that captures subject area expert knowledge to leverage inherent parallelism in systems in the following ways:

- Data structures are separated from instructions to track which instruction sets share what data. This is used to determine independence and thus the potential for concurrent processing at run-time.
- Model connectivity (independence) can be inspected visually to determine if the inherent parallelism of a physical system is properly represented. Models need not be changed to move from a single processor to parallel processor hardware architectures.
- Knowledge of the architectural parallelism is stored within the system and used during run time to allocate processors to processes in a maximally efficient way.

Keywords: Modeling, Simulation, Parallel Processing, Discrete Event, Visual Architecture, High Level Language

1. 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.

[*] bill@predictsys.com; phone 1-732-449-6800; fax 732-449-0897; www.predictsys.com

1.1 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.

2. 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.

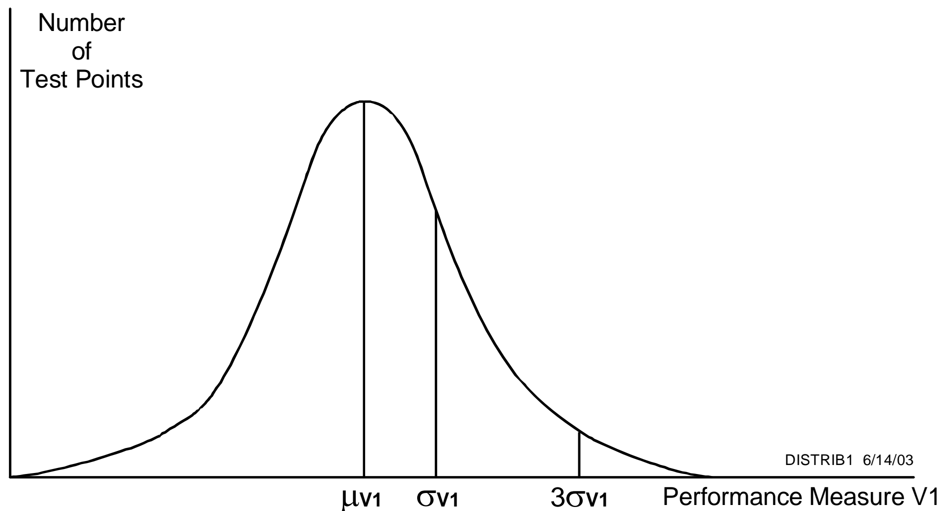

Figure 1. Illustration of the distribution of test results for performance measure V1.

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.

2.1 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.

2.2 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.

2.3 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.

3. 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.

3.1 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 M1, M2, ..., MN 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.

3.2 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].

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. The scope of a resource, and thus every attribute, is determined by the architecture, not the language.

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 size 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 size 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.

4.1 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.

4.2 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.

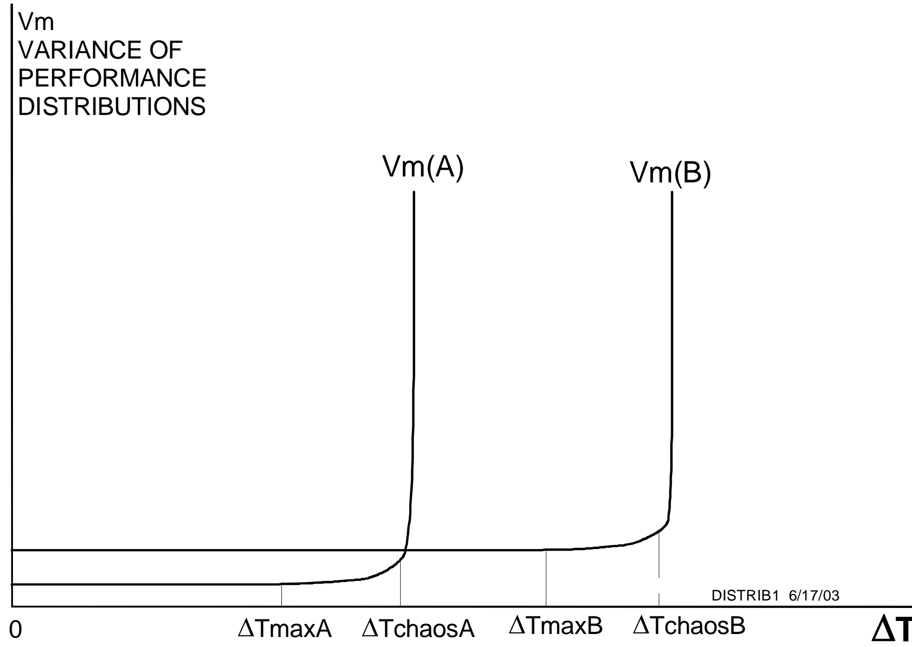

Figure 2. Illustration of the selection of ΔTmax.

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.

4.3 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.

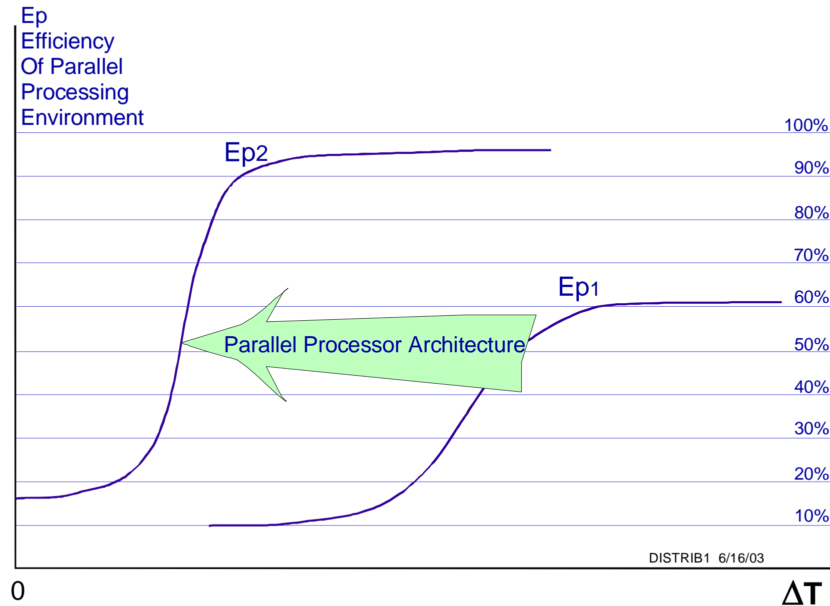

Figure 3. Illustration of parallel processor efficiency versus ΔTmax.

4.4 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.

4.5 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).

4.6 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 ΔT_{max}. 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 ΔT_{max}. For this example, Ep2 clearly achieves the goal.

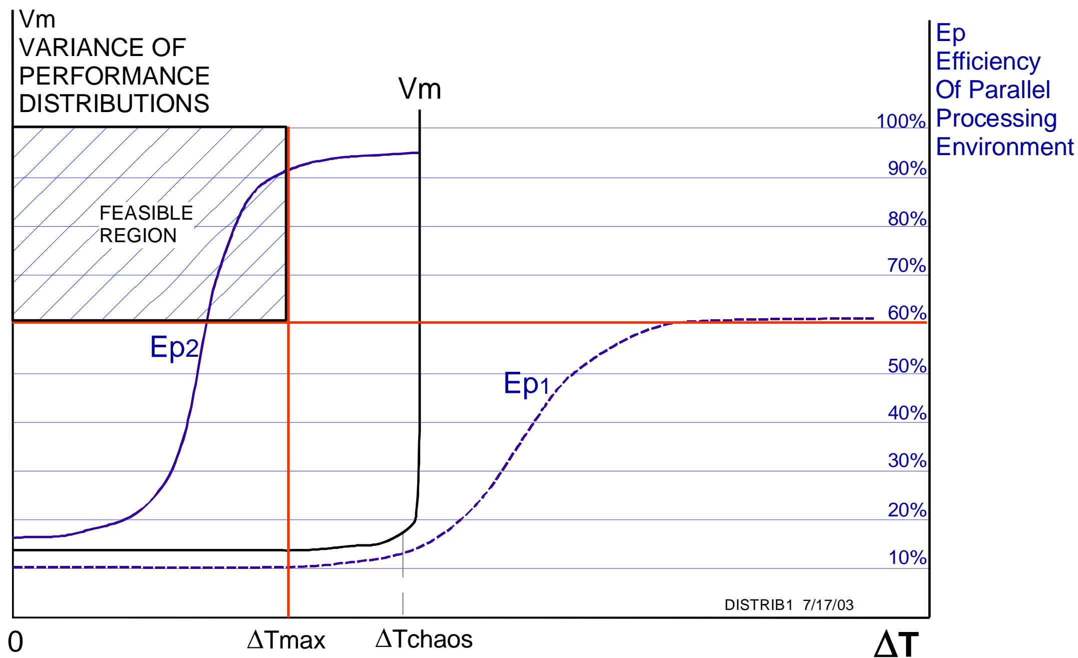

Figure 4. Relative effects of parallel processor architecture on validity.

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.

5. 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.

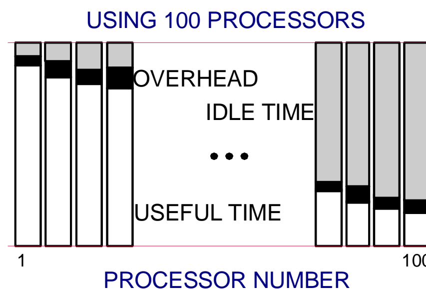

Figure 5. Ordered processor loading histogram.

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.

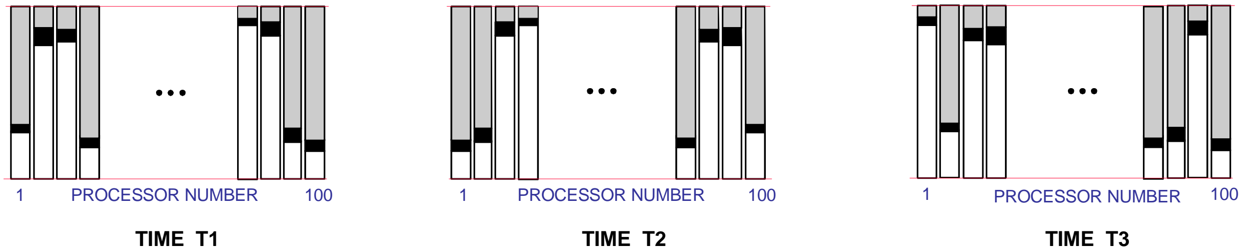

Figure 6. Changes in processor loading over time.

5.1 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.

5.2 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.

5.3 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.

6. 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. Conversely, well-designed independent model architectures will not benefit from run-time software that does not take advantage of that knowledge. Finally, high latency hardware architectures may render good model and run-time architectures as infeasible solutions.

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.