UNDERSTANDING INHERENT PARALLELISM

William C. Cave† & Robert E. Wassmer† - March 11, 2008

INTRODUCTION

The concept of *inherent parallelism* is used frequently in the literature addressing parallel computation. Typically it is in the context of estimating speed multipliers that may be obtained from a parallel processor. Various authors have described the pitfalls encountered when attempting to derive formulas for such estimates, see [2]. Disagreements often cite improper measures of parallelism, or serial versus parallel parts of a program.

Most papers addressing speed multipliers cite Amdahl's 1967 presentation, [1], and proceed to support or deny his conclusions. "Amdahl's Law", apparently derived by others from his presentation, defines a speedup multiplier one may obtain from a parallel processor that depends upon the ratio of the serial part to the parallel part of a program. Various authors have argued the validity of Amdahl's law, questioning the definitions of serial and parallel parts in coming up with their own laws, see [9] and [14]. But none of these "laws" take into account many factors that affect measured speed multipliers, e.g., a limit based upon number of independent modules, regardless of the number of processors, see [4]. Basic issues on single processor versus parallel processor software architectures are generally mistreated. These questions and issues have been further addressed in [5].

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

ANALYSIS OF THE ISSUES

As used in the literature, the term "inherent parallelism" appears to imply the property of a system that allows one to decompose a software task into elements that can run concurrently on separate processors. As used here, a software task is an independent executable in the context defined in [6]. Upon further reflection, one sees misunderstandings of whether the property of inherent parallelism pertains to a system, or the code that implements it. To understand this, consider that parallel processing has a long history of being used in large scale simulations of physical systems. In that environment, people use the term "codes" to imply the simulations they have built. Because of the complexity of these codes, and the time it has taken to debug them (typically years), changing these codes is considered anathema. When porting these codes to different parallel processors, it is the inherent parallelism of the code that is often referenced, not the system that the code represents.

† *The authors are with Visual Software International (www.VisiSoft.US)*

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

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

To gain perspective on the problem, we will consider some limiting cases. We start with systems represented by mathematical models, many of which use sets of differential equations resulting in large 2 or 3 dimensional arrays of the type to which Amdahl referred. In such problems one may have to invert large sparse matrices thousands of times. Known solutions to this problem are described in [3] and [10], wherein one generates a symbolic solution in code. This approach minimizes operation counts (e.g., add, multiply, etc.) and eliminates loops. Typically every instruction depends upon the prior one - implying that no instructions can run concurrently. Such code has been designed to run very fast on a single processor and has virtually no inherent parallelism. However, the systems (e.g., electrical networks) represented by this approach may have substantial inherent parallelism. Most of the physical elements operate concurrently with many others. Other approaches to modeling these systems, e.g., using discrete event simulation, generate code that is entirely different, operating much like the physical system when run on a parallel processor. These codes may have a high degree of parallelism.

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

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

CONCURRENT PROCESSING

We start by considering a basic set of definitions using the simplified representation illustrated in Figure 1. This figure shows utilization of N processors by a software system running on a parallel processor during a sample period (ΔT). The green area is processor time doing useful work. The black area is processor time spent in overhead functions. The pink area represents time when the processor was idle. The blue area is time after which that processor terminates its use (note that processor 1 happens to terminate its participation in this sample period). During the course of ΔT, useful work, overhead, and idle time may be interspersed, as modules on one processor communicate with modules on another. If none of the green areas overlap in time, then no useful work is done concurrently, and using parallel processors will not shorten the run-time. This does not imply there is no inherent parallelism in the system. It merely implies that the approach to decomposition of the software (the software architecture) contributes no positive speed multiplier when run on a parallel processor.

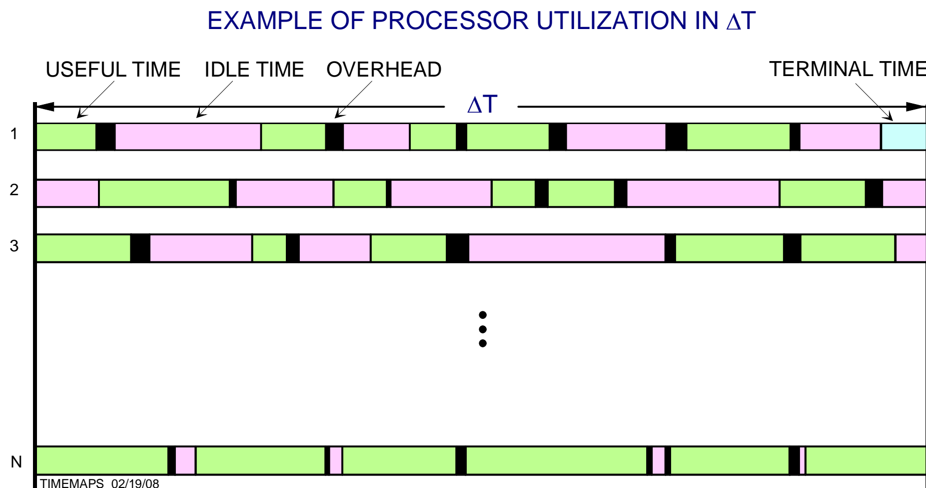

Figure 1. Example of processor utilization in ΔT.

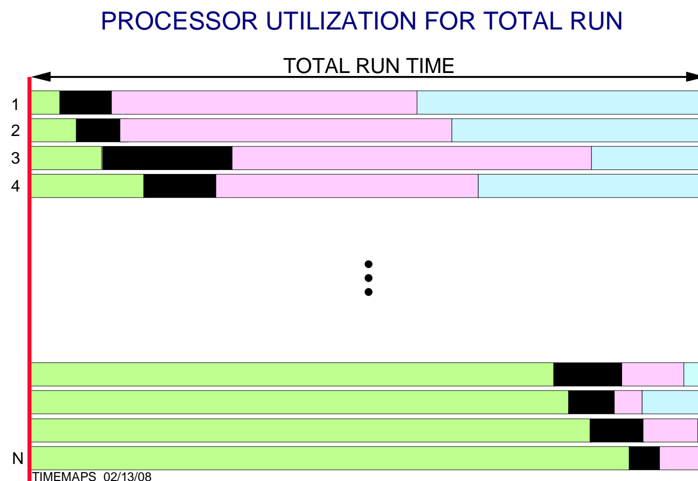

Figure 2. Grouped and ordered processor utilization for entire run.

Figure 2 shows the relative amounts of processor utilization, overhead, idle time, and termination time grouped together for each processor and ordered by utilization from top to bottom. Clearly the amount of useful work at the bottom of this chart implies significant overlap had to occur, whereas at the top of the chart, one would not expect much overlap if any.

ESTIMATING PROCESSOR UTILIZATION EFFICIENCY

If one plots a curve of the useful time spent on each processor (green bars shown in Figure 2), one expects different outcomes for different systems, as well as for different architectures of a given system. Figure 3 illustrates different possible outcomes using different shades of green. The lightest shade of green has the least useful overlap, the middle shade is in between, and the darkest shade has the most.

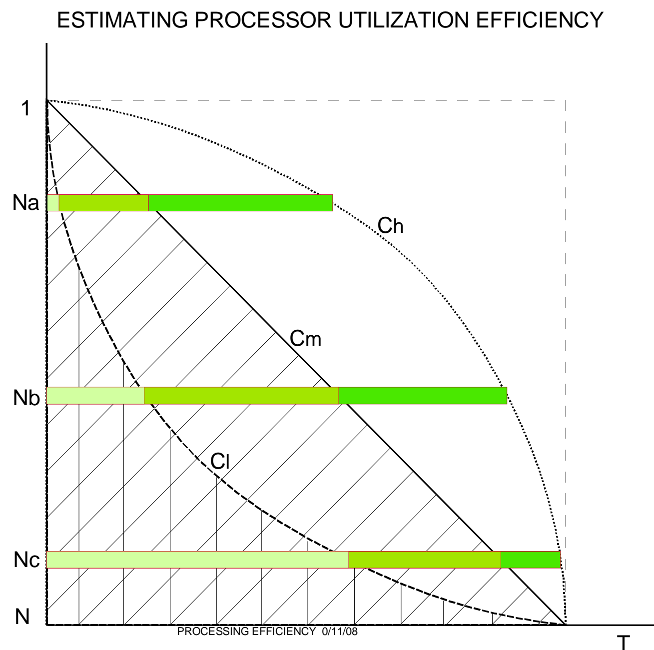

Figure 3. Grouped and ordered useful processor time for entire run.

If the middle shade follows the straight line, Cm, the area under the curve yields a 50% overlap across processors, implying 50% processor utilization efficiency. This would yield a speed multiplier of approximately N/2. In the limiting case, as the curve governing the light green shade, Cl, falls into the origin, the processor utilization efficiency approaches zero. Similarly, as the curve governing the dark green shade, Ch, approaches the upper right hand corner, the processor utilization efficiency approaches 1, and the speed multiplier approaches N. In this case the inherent parallelism in the system can be translated into an architecture such that N processors can be used with approximately 100% useful overlap. In general, the area under these curves represents the corresponding processor utilization efficiency when normalized to the range [0, 1].

We note that processor utilization efficiency may be increased by shifting all work from one processor to an under utilized processor, decreasing the number of processors used. This will generally increase overall run-time if there was useful overlap between the processors.

Alternatively, one may shift work from an over utilized processor to an unused processor, decreasing processor utilization efficiency but decreasing run-time if there is useful overlap between these processors.

Based upon the above analysis, processor utilization efficiency of 1 implies no idle time, terminal time, or overhead. This is typically approached only by embarrassingly parallel systems and solutions. Our concern here is understanding how to minimize these effects for systems that are only partially independent, see [8].

Figure 3 provides a closer look at processor utilization. Useful time includes the time spent performing overhead functions that would also be performed on a single processor. Idle time is broken out into excess time spent waiting for inter-processor data transfers and control transfers. Overhead time is broken out into excess time spent reallocating resources (e.g., for processor assignment, load balancing, etc.) and excess time spent for other parallel processing overhead.

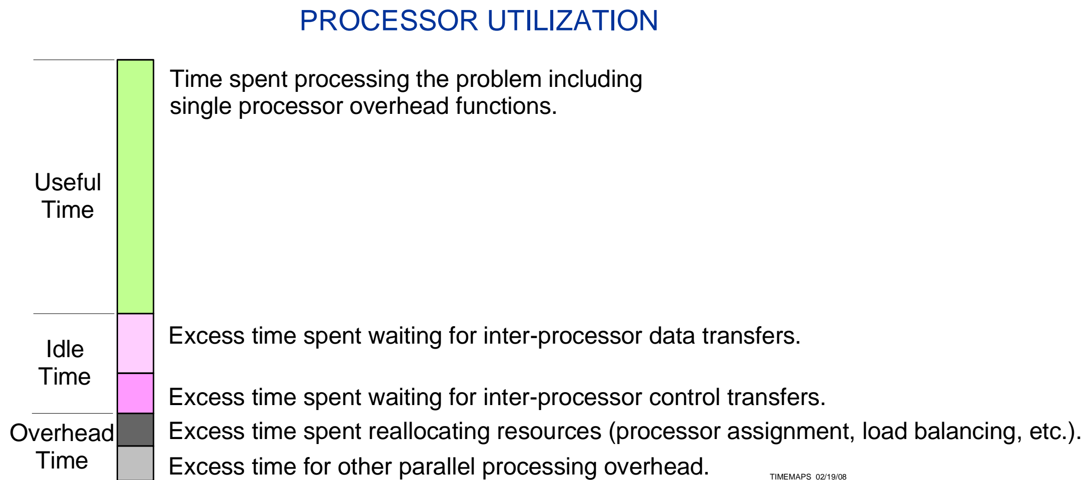

Figure 3. Closer look at processor utilization.

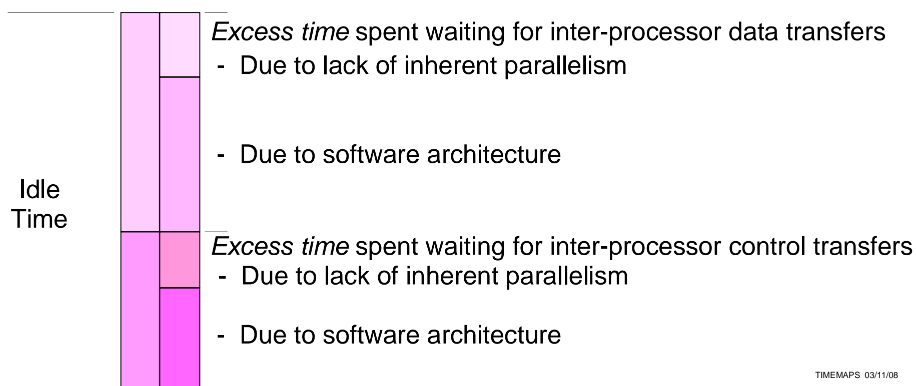

Figure 4. Closer look at idle time.

Overhead time is affected predominately by hardware and OS architectures, see [4]. Figure 4 provides an even closer look at the idle time. Waiting for both inter-processor data transfers and control transfers can be caused by a lack of inherent parallelism, or by the software architecture. These two items are considered the important factors to be dealt with when trying to maximize speed multipliers using parallel processing. We deal with each of these next.

INHERENT PARALLELISM

Whereas other authors are interested in parallelism contained in existing codes, see [11], our interest is in developing software architectures that take maximum advantage of the inherent parallelism of a system. As shown in the charts above, without useful time overlap, there is no gain in speed. Idle time is likely to be the major cause of reduction in overlap of useful time on parallel processors. As stated in Figure 4, idle time is caused by the lack of inherent parallelism in a system or the lack of a software architecture that takes advantage of the inherent parallelism.

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

Discrete event simulation provides significant insight into this problem, see [8]. When building model architectures to support simulations to be run on parallel processors, one looks for concurrency in the system being simulated. By this we imply that elements of the system must operate independently of each other. Models built using an architecture that preserves the concurrency properties of the system provide for concurrency when run on separate parallel processors. In both cases, concurrency of operations can be achieved only when those operations are independent. In addition, the scenario driving the simulation will affect the potential overlap in useful processing time, and the corresponding potential to speed up the simulation. When modeling communication systems, message traffic overlap in different parts of the system will define the inherent parallelism. Again, one must consider the worst case design constraints.

Recognition of inherent parallelism may be described using a state space framework. In particular, simulation models and software may be characterized in terms of vectors and transformations using a Generalized State Space Framework, see [7]. In this framework, a model consists of state vectors (data) and transformations (instructions). This leads to a visualization of the connectivity properties of the transformations as shown in Figure 5.

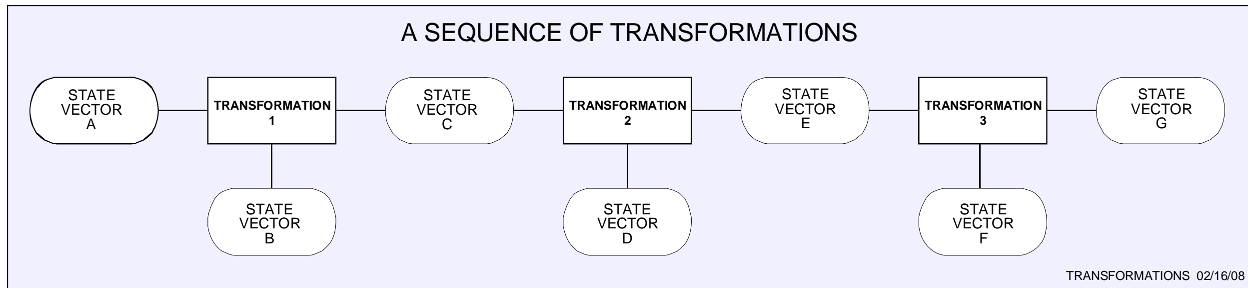

Figure 5. State vectors and transformations.

Each transformation has a dedicated state vector and a shared state vector. Transformation-1 has state vector A as input, shares state vector C with transformation 2, and has state vector B for dedicated use. When interactions are directly coupled, the actions of one transformation immediately affect the state of the other. But, no matter how small, there is a finite time delay between the operation of transformation 1 and the response seen by transformation 3.

Transformation 1 is directly coupled to transformation 2 which is directly coupled to transformation 3. However, transformations 1 and 3 can operate concurrently since they share no state vector directly. Transformation 2 can operate when transformations 1 and 3 are idle.

As shown by various authors, any physical system may be modeled to within any measurable degree of accuracy using the state-space framework, see for example [12] and [15]. From this we may derive that the inherent parallelism in a system may be modeled by representing the operations of a system using the state-space framework as shown in Figure 5. As shown in [7], these models need not be mathematical models, but may be discrete event simulation models or general pieces of software.

The critical property determining concurrency of operations, and thus inherent parallelism, is that of independence as described above. This implies that two transformations are independent when they share no state information. The separation of state information (data) from transformations (instructions) provides the basis for the Separation Principle for software as described in [5], [7], and [8]. The Separation Principle yields drawings of software architecture, similar to that in Figure 5, providing direct visualization of module/model independence.

Theoretically, one can determine the inherent parallelism in a system by creating a model of it where the elements are represented by state vectors and transformations similar to that in Figure 5. As shown in [5], a proper choice of state vectors will maximize the independence of models, taking maximum advantage of the inherent parallelism, and optimizing concurrency.

A similar (if not identical) model may be developed for use as a discrete event simulation. If the simulation uses the same architectural framework as the state space model, then it will also contain the same inherent parallelism as that model, and theoretically, the system itself.

TEMPORAL INDEPENDENCE AND DECOMPOSITION

As indicated above, models may be considered independent during a time period that is sufficiently small to provide a valid representation of the real system. If this time period is large enough, then sufficient processing may be done in parallel. If it is too small, then the effects of a module in one processor on that in another processor may be too tightly coupled to benefit from running on separate processors (the results will be valid, but the speed gains insufficient). This problem occurs in various modeling problems as described below. We must answer the question: When can a system be decomposed into separate models so as to provide effective speed improvements on a parallel processor while maintaining validity? To answer this question, we must address validity.

ENSURING VALIDITY

Test results can be presented in many ways. We use M to denote a generalized vector of values measuring the properties of a system under test. 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 as illustrated in Figure 6.

When field or laboratory testing is prohibitive or expensive, one typically resorts to simulation. One must then assess the cost of obtaining valid results from a simulation. When running a single processor simulation, one will get the same results from every run unless it is taking in data generated in real time. To provide a more accurate view of results, values of parameters that may vary in real operations are drawn randomly from predetermined distributions representing known or anticipated variations. These distributions, also illustrated in Figure 6, are used to analyze validity of simulation results, see [4].

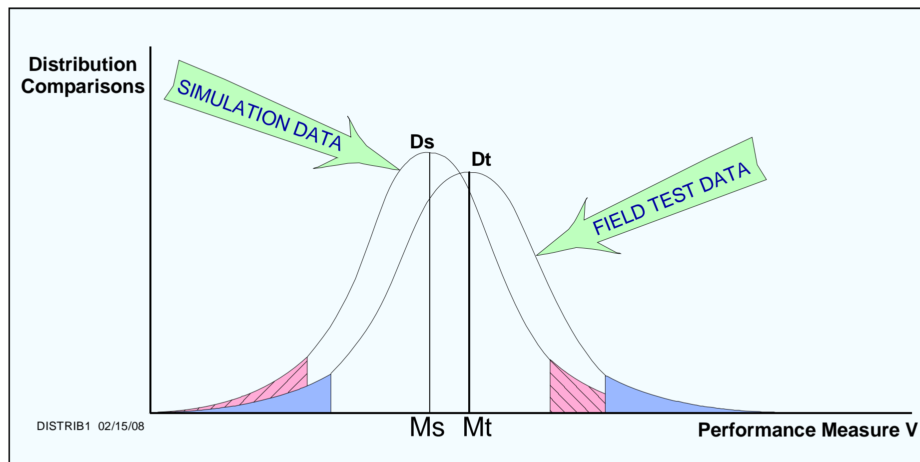

Figure 6. Illustration of the distribution of test results for performance measure V.

To analyze the effects of variations, one typically runs Monte Carlo simulations, whereby the simulation is run a sufficient number of times, each with a different random number seed, to produce a distribution of results. Then one can compare the distribution of results from the simulation, Ds, to that of a valid test set, Dt. 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 valid distributions for the measurement vector. We are concerned with obtaining sufficiently valid results when moving from a single processor to a parallel processor environment.

Validating Parallel Processor Simulations

Based on the above, we will start 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 with the assumption that the values will be 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 made regarding the coherency of the data. In software, this problem is 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. In live field tests, ensuring such sequences always occur the same way is typically very difficult if not impossible. This is why it is common for single processor distributions to be more narrow than the corresponding field tests.

To validate the results of a simulation, one must compare the distributions of the measurement vector from the parallel processor simulation to that of a valid test set, or valid single processor simulation (single thread implied). Validity can be achieved without having the same process sequence. The process sequence is typically varied in the single processor case simply by varying the random number seed. As stated above, maintaining a strict sequence 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.

In a parallel processor simulation using the General Simulation System (GSS), one can ensure that the simulation clocks on each processor do not differ by more than ΔT, a parameter specified by the modeler, see [4] and [16]. We note that, using GSS, the 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. This is a key factor in obtaining higher speed multipliers from parallel processing, reducing idle time of processors waiting for clock synchronization. Figure 7 shows the effects of increasing ΔT in two different simulations, A and B. As ΔT is increased, a point is reached, ΔTchaos, where the simulation becomes chaotic as shown by the rapid increase in variance of the distributions at that point.

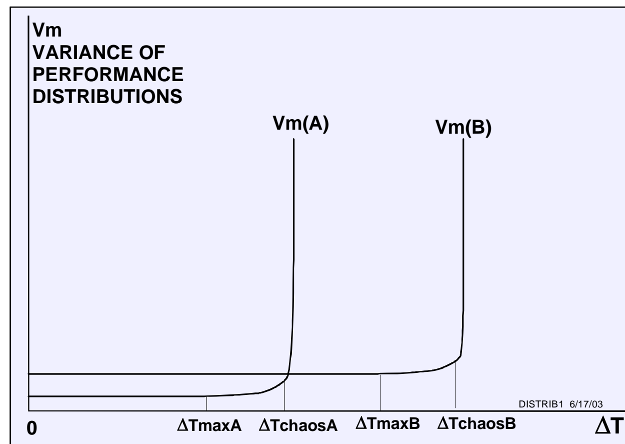

Figure 7. Illustration of the distribution of test results for performance measure V.

The motivation for increasing ΔT is improvement in parallel processing efficiency and the resulting speed multiplier that is obtained. References [4] and [16] also show how the architecture of a parallel processor affects the speed multipliers.

Modeling Dynamic Systems

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

In the case of linear systems, one can determine simple models (equivalent circuits) that present an equivalent of the input to the next stage. This allows large models to be broken into smaller decoupled models. In these cases, ΔT may be determined by the amount of detail one wants to see in the output response, becoming in effect a sampling rate. The minimum sampling rate, ΔT, is typically calculated from the shortest time constant in the system and Shannon's theory, [13]. In general, separate models will have different time constants, requiring different sampling rates. One must use the fastest sampling rate of models that interact.

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

Given that models may be split across processors to run concurrently, they will run until they exceed the selected ΔT synchronization point. At that point, all must wait for the slowest one. This causes idle time on the rest, reducing processor efficiency. Processor efficiency may be raised by putting multiple "slow" models on a single processor, using fewer processors. However, if there are a sufficient number of processors, using them less efficiently may be the easiest and certainly fastest approach.

SUMMARY

Upon analyzing the issues relative to inherent parallelism in systems, we have described properties that aid in designing software systems or simulations for parallel processors. The principle property is that of the independence of operations of the physical system. If these independent operations can be modeled using a state-space framework, then the state vectors separating the independent transformations, and the transformations themselves, can be built as software modules or models of the system. Because of their inherent property of independence, these modules or models can be run concurrently on parallel processors.

REFERENCES

[1] Amdahl, G.M., Validity of the single-processor approach to achieving large scale computing capabilities, Proceedings, AFIPS SJCC, Reston, VA, 1967 - (transcribed by Guihai Chen).

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

[3] Berry, R.D., An Optimal Ordering of Electronic Circuit Equations for a Sparse Matrix Solution, IEEE Transactions on Circuit Theory, Vol 18, Issue 1, Jan. 1971.

[4] Cave, W., et.al, The Effects of Parallel Processing Architectures on Discrete Event Simulation, Proceedings: SPIE Defense & Security Symposium, Mar/Apr 2005, Orlando, FL.†

[5] Cave, W., et.al, Getting Closer to the Machine, Visual Software International Technical Report, Mar 2007, Spring Lake, NJ.†

[6] Cave, W., et.al, Estimating Parallel Processing Speed Multipliers, Visual Software International Technical Report, Oct 2007, Spring Lake, NJ.†

[7] Cave, W., A Generalized State-Space Framework for Software, Visual Software International Technical Report, Nov 2007, Spring Lake, NJ.†

[8] Cave, W., Using Parallel Processors, Visual Software International Technical Report, Feb 2008, Spring Lake, NJ.†

[9] Gustafson, J.L., Reevaluating Amdahl's Law, CAACM, 31(5), May 1988.

[10] Hactel, G.D. et al, The Sparse Tableau Approach To Network Analysis And Design, IEEE Transactions on Circuit Theory, Jan 1971, pp 101.

[11] Kumar, M., Measuring Parallelism in Computation-Intensive Scientific/Enineering Applications, IEEE Transactions on Computers, Vol 37, Issue 9, Sep 1988.

[12] Schweppe, F., *Uncertain Dynamic Systems*, Prentice Hall, Englewood, NJ, 1973.

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

[14] Shi, Y., Reevaluating Amdahl's Law and Gustafson's Law, Temple Un., Oct 1996.

[15] Zadeh, L.A. and Desoer, C.A., *Linear System Theory: The State Space Approach*, McGraw-Hill, NY 1963.

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

† These references may be found on the VisiSoft.US website.