Estimating Parallel Processing Speed Multipliers (10/15/07)

William C. Cave
Visual Software International
bill@predictsys.com

Robert E Wassmer
Visual Software International
bob@predictsys.com

ABSTRACT

Speed multipliers to be gained from using a parallel processor depend upon various factors including the inherent parallelism in the system itself, the hardware architecture of the machine, and an OS with flexible facilities to allocate and assign processor and memory resources.

Given these factors, one still requires a software development environment that makes it easy to design architectures of independent modules that can run concurrently, with minimum communication, therefore reducing overhead and idle time. It also requires that a run-time environment be generated with knowledge of the module independence (i.e., the software architecture), so that modules are allocated to processors, and threads assigned within those independent modules, in a way that takes maximum advantage of parallel processor resources.

Categories and Subject Descriptors

Parallel Processing, Software Speed, Software Independence, Software Module, Module Independence, Software Architecture

General Terms

Design, Architecture, Economics

Keywords

Parallel Processing Speed, Software Architecture, Amdahl's law

1. INTRODUCTION

Estimation of potential speed multipliers to be gained from parallel processing began in the late 1950's. Since then, large investments have been made in different hardware architectures. Except for special problems, most have provided small returns. Estimating speed multipliers for time-critical applications is a difficult problem involving many factors and viewpoints, see [6] and [3]. One view is that of computer manufacturers looking at potential markets and corresponding software applications that affect sales, see [1]. Other viewpoints, e.g., those dealing with scientific applications involving large simulations and extreme speed constraints, are also well represented, see [5]. Software investments have been restricted to "parallel constructs" added to existing languages.

Commercial applications involving a large single task are generally built around a centralized database, making distribution to many processors unattractive. Except for client-server transaction processing, most of these applications are not severely constrained to the real time clock. Transaction processing applications are already parallelized, distributed over large numbers of client-server platforms. However, these applications are "embarrassingly parallel," running on separate platforms, each with their own operating system (OS).

Scientific applications have traditionally used mathematical frameworks, e.g., differential or difference equations, to solve dynamic problems. Problem solutions using this type of mathematical framework are typically solved using matrix calculations. These abstractions of the physical world produce computations centered around large matrix databases shared by all of the computational parts. Unless a database can be distributed into independent pieces with minimal sharing, it is difficult to map the software architecture onto a parallel processor to produce worthwhile speed multipliers.

Over the past three decades, processor clock rates have followed the Moore's curve, doubling every 18 months. Every three years, single processor tasks ran 4 times faster, making the parallel processor case difficult to justify for all but special applications. In the past few years, clock rates have flattened. The Moore's curve now contributes only to larger cache memories and wider instruction sets. Although these improvements support speed increases, the contributions are indirect and small compared to the prior 25 years.

In scientific applications, large simulations are moving toward discrete event solution approaches. The principal motivation is the ability to develop models that follow the physical design of a system, instead of mathematical abstractions. This simplifies the modeling while improving both the accuracy of representations and the ability to add resolution as needed. Subject area experts can participate in a modeling process that does not require special programming expertise. More importantly, the move from abstract representations to those following the design of a physical system contributes directly to the independence of modules. This is because the systems being modeled are designed to maximize independence of physical modules for easy replacement. This approach provides a software path to parallel processing.

We note that the analysis provided here is based upon the unsaturated case (number of processors available equals or exceeds number of independent modules). The saturated case is more complex, depending upon migration of modules to balance loading and having independent modules share the same processor. The approach prescribed here still applies, but the analysis is beyond the scope of this paper. However, it is the experience of the authors that good architectures can be produced using large modules whose numbers remain within the limit on number of processors.

2. PRIOR ANALYSES

The literature on estimating potential speed gains using parallel processors is somewhat divided between those claiming significant improvements and those claiming it is likely not worth the cost. Most papers cite Amdahl's 1967 paper, [1], and proceed to support or deny his conclusions. "Amdahl's Law", apparently derived by others from his paper, defines a Speed Multiplier, SMA, when using a Single Operating System (SOS) parallel processor using the mathematical formula:

$$SMA = \frac{1}{(1-p) + \frac{p}{n}} = \frac{n}{n(1-p) + p}$$

where p is defined as the percent parallelism and n is number of processors.

The following are examples. When p is 75%, the Speed Multiplier approaches 4 as n gets large. If p is 95%, and n is 100, the Speed Multiplier is 16.8. Further examples are shown in the plot in Figure 1.

Various authors have argued Amdahl's law, questioning the definition of parallelism, p, see [6]. Generally, a system with 99% parallelism is considered embarrassingly parallel. However, with 100 processors, the Amdahl Speed Multiplier is 50, when one would expect a figure in the 90's. In fact, Gustafson's law shows that speed multipliers greater than 1000 have been obtained from 1024 processors, see [5]. But neither of these "laws" take into account many factors that affect the multiplier, 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.

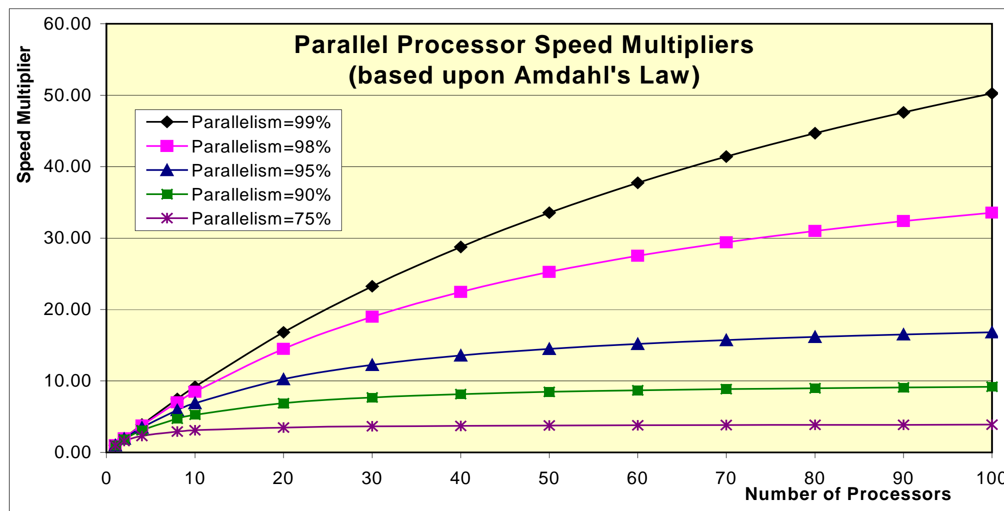

Figure 1. Parallel processor speed multipliers based upon Amdahl's Law.

3. ANALYSIS OF THE ISSUES

Inherent parallelism implies the ability to decompose a software task into modules that can run concurrently on separate processors. The limiting cases are: (1) 0%, i.e., every instruction depends upon the prior one - implying that no instructions can be run concurrently; and (2) 100% - known as the embarrassingly parallel case. The usual 100% example is Monte Carlo analysis using N simulations, each with a different random number seed, running concurrently on N processors. But even in the embarrassingly parallel case, the maximum speed multiplier remains at N, even when the number of processors is increased. Also, Amdahl's Law produces a dramatic jump between the 99% case and the 100% case. This jump appears at odds with basic principles as addressed by Gustafson, [5]. Finally, we are concerned with useful measures that fairly represent what's "best" in terms of a set of economic choices.

There are a number of critical issues to be considered when analyzing the potential speed multiplier expected using a SOS parallel processor. Our goals are to develop models for estimating this multiplier so that: (1) approaches to building software for parallel processors can be compared theoretically; and (2) models can be calibrated and validated by experiment. We start by considering a basic set of definitions using the simplified representation illustrated in Figure 2.

The left-hand side of Figure 2 shows utilization of N processors by a software system running on a SOS 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. 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.

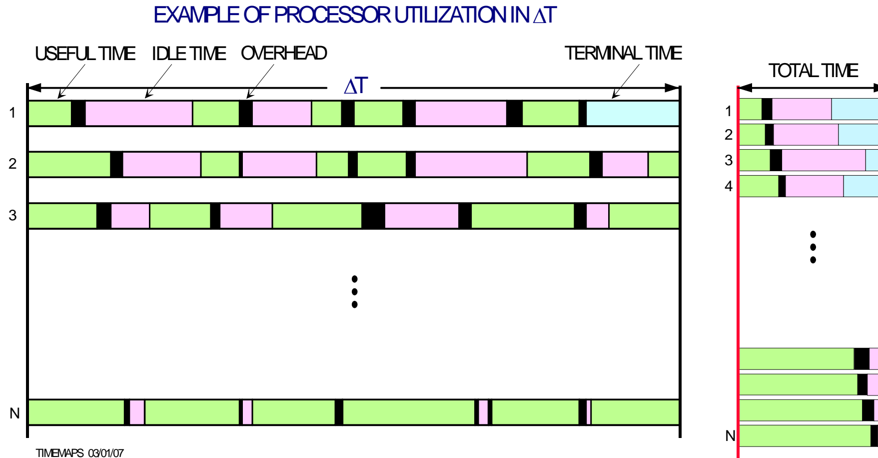

Figure 2. Example of parallel processor utilization.

On the right-hand side of Figure 2 is a chart of rows (again one for each processor) containing representative utilization grouped and ordered by useful time, overhead, idle time, and terminal time. The period represents the total run-time. The rows are also ordered by increasing useful time. It is apparent in this example that the decomposition has taken some advantage of inherent parallelism since there is reasonable overlap in useful time at the bottom of the chart.

Note that considerable idle time exists in the top part of the right chart, indicating that modules on those processors had to wait to be invoked or wait for input from another processor, implying that smaller percentages of concurrency were achieved. This could have been due to a lack of inherent parallelism, or due to the software architecture. We also note that different scenarios driving a software system or simulation may have a significant effect on these outcomes.

The following are definitions of useful time, overhead time, and idle time. They are independent of the software and hardware architectures.

Useful Time - That spent running instructions to perform the functions of the task that would be required on a single processor (TU).

Overhead Time - That spent running excess management or communications instructions on a parallel processor that are part of the task, as well as instructions not in the task (OS instructions) (TOH). OS instructions include those for swapping instructions in the task to be processed by a particular processor, those used to page memory required by task instructions on a particular processor, as well as those for other OS overhead.

Idle Time - That spent running no instructions (TI) - with the processor idling in a wait state - waiting to be invoked or waiting for communications.

These definitions apply to single or parallel processors except as noted. They assume that no other task is running on the processor (otherwise, results will depend upon what else is running).

4. PARALLELISM, ARCHITECTURE AND DECOMPOSITION

Parallelism implies that a software architecture can be produced that decomposes the system into modules such that modules can be designated as independent, implying that they may run concurrently with other modules in the system when they are invoked. The decision to invoke a module at run-time is based upon the application requirements and software design. We note that the software architecture for a single processor may be different from that for a parallel processor. For example, copies of memory in each module may be used to gain speed in a parallel processor, whereas using more memory may cause more paging on a single processor. Such factors must be considered to compare speeds fairly, see [2].

The *order* of a decomposition is equal to the number of independent modules. Decomposition of order M implies that the architecture has decomposed the application into M independent modules that can run concurrently with each other as well as the rest of the system.

When using n processors in parallel, the speed multiplier, Sn, is the ratio of measured values of run-time on a single processor (R1), and run-time on n processors (Rn):

$$S_n = \frac{R_1}{R_n}$$

The percent parallelism, P, achieved by a software architecture running on a machine with up to N processors is measured as follows.

$$P = \underset{n=1 \rightarrow N}{\mathrm{MAX}} \left[\frac{S_n}{n} \right]$$

We note that these measures must be obtained by experiment, and are independent of whether the software or hardware architecture is the same for both the single processor case and the parallel processor case. If minimum run-time for a single processor is achieved using a different software architecture than that for the parallel processor case, then different architectures should be used to produce a fair value for the speed multiplier. Likewise, one may use different hardware architectures; dependence of the speed multiplier upon hardware is described elsewhere, [3]. As indicated above, these outcomes may be

scenario dependent, requiring measures of statistical distributions for validity, see [3]. We can now define the inherent parallelism of a system (and scenario) as the maximum percent parallelism, PMAX, that can be achieved theoretically by the best hardware and software architectures for both the single and parallel processor cases. We now describe ways to estimate these measures based upon models of hardware and software architectures and factors that affect them.

5. ESTIMATING SPEED MULTIPLIERS

Given a software architecture that decomposes a system into independent modules, a module may take the same amount of useful processor time on a separate processor as it would running in the single processor case. However, if the hardware or software architecture in the parallel processor case is different from that of the single processor, then the useful running time will likely change. For example, a table may be used to store information for hundreds of instances of an entity being simulated on a single processor and common data for each instance may occur once. On a parallel processor, the data for each instance may be on a separate processor, eliminating the table, but the common data will be repeated with each instance.

If the table is large, then paging may be required across multiple memory boundaries on a single processor, but not on a parallel processor since the table size may be cut by the number of instances. Common data is usually small compared to table sizes, but that also depends on the application. The point is that a decomposition that minimizes the run-time for a single processor will likely differ from that for a parallel processor. Also, cache sizes local to a processor may be the same for each parallel processor as for a single processor. Or one may select a single processor with a much greater cache size to reduce the swapping and paging overhead from that of one of the parallel processors. Regardless of the differences in hardware, given that communication between processors is treated separately, swapping and paging within a processor on a parallel machine may be reduced dramatically from that on a single processor.

Overhead time may change also. For example, when modules on different processors are communicating, overhead will accrue that is not needed on a single processor. In addition, excess overhead time as well as idle time may accrue waiting for return communications. This time will effectively reduce the measure of parallelism. However, if a module requires a large database, and that database can reside in the cache of its designated processor with no paging, then overhead time may be reduced when it runs on a separate processor.

For each independent module, one must estimate the useful time, overhead time, and idle time for both the single processor case and the N processor case - where N equals the number of modules. This may be accomplished by estimating the single processor case, and then treating the parallel processor case as a normalized percentage (increase/decrease) of the single processor case. Communication between modules in different processors will accrue additional overhead as well as idle time in the parallel processor case. Thus overhead must be broken into parts: (1) time required for communication between processors, TC; (2) time required for swapping, TS; and (3) time required for paging, TP. For the mth module, total runtime overhead is given by

$$T_{OHm} = T_{Cm} + T_{Sm} + T_{Pm}$$

Both communication between processors and paging depend upon the software architecture as well as the parallel processor facilities. Communications will depend upon the size of data transfers, and the memory boundary crossing delays between processors. Paging will depend upon the memory size required for each module, the amount of local cache next to each processor, and the delays associated with each memory boundary crossing.

The major factor affecting the speed multiplier is the ability to achieve overlap of useful times as illustrated in Figure 2. To estimate this, one must effectively estimate the useful time overlap between each module. This is not a simple process. One must consider that an overlap of 50% of the modules during a given time period represents 50% parallelism being taken advantage of during that time period. This must be done for all time periods. It is possible to capture this data by instrumenting the software in a way that is virtually nonintrusive.

To estimate the run-time for a single processor, (RSE), useful time and overhead must be summed for each module, accounting for swapping and paging overhead.

$$R_{SE} = \sum_{m=1}^{M} \left[T_{Um} + T_{OHm} \right]$$

where m represents the m-th module.

For the parallel processor case, the estimated run-time, (RPE), will equal that of the module with the largest sum of useful and overhead time (recalculated), and idle time.

$$R_{PE} = \underset{m=1 \rightarrow M}{\mathrm{MAX}} \left[T_{Um} + T_{OHm} + T_{Im} \right]$$

Since idle time and overhead may play significant roles, one must perform this estimate for each module to determine the total time. Modern parallel processors collect data that can help to estimate these times. Without such data, estimates may be very difficult if not intractable. What is important is to analyze this data to understand the underlying technology for improving software architectures.

6. THE IMPORTANCE OF SOFTWARE ARCHITECTURE

If an architecture produces a number of independent modules where threads do not cross module boundaries, and the number of independent modules does not exceed the number of processors, then swapping overhead may be reduced to zero if each module remains by itself on a processor. This requires a run-time system that has the information on module independence to put all threads in an independent module on the same processor, see [4]. It also implies OS facilities for controlling processor allocation and assignment based upon requests from the run-time system.

If the interfaces between modules are minimized, then paging is also minimized. With a good software architecture, paging overhead for a module on a parallel processor may be significantly lower than that on a single processor if not nil. If there is a very high degree of inherent parallelism, and a good software architecture to take advantage of it, it is conceivable that the speed multiplier could be greater than the number of processors. This would result from excessive swapping and

paging overhead incurred on a single processor that disappears when a large number of modules are spread onto separate parallel processors. To achieve this, one must have a development environment that aids in the creation of large independent modules, and a run-time environment that is provided with knowledge of the software architecture so it can be taken advantage of at run-time. If the run-time environment also contains facilities that ensure data coherency between modules on separate processors, then paging will likely be nil for those modules, being replaced by transfers between data structures at the application level.

7. SUMMARY

The speed multipliers to be gained from using a parallel processor depend upon a number of factors that may be evaluated prior to making an implementation investment. First is the inherent parallelism of the system. This determines the potential for useful processing to occur concurrently (useful time overlap) on a parallel processor. Second, the hardware architecture must support the ability to access memory local to each processor concurrently as well as minimize the time for transfers between processors. Third, the OS supporting the hardware must provide facilities to allocate and assign processor resources by a run-time system that has been optimized to use knowledge of the software architecture.

Given that the above criteria are met, the most difficult hurdles to date have been the ability to: (1) effectively decompose a software application with inherent parallelism; and (2) effectively use the knowledge of that decomposition at run-time. This requires a software development environment that makes it easy to develop architectures of independent modules with minimized communication between them, reducing overhead and idle time. It also requires that a run-time environment be generated with knowledge of the module independence (the software architecture), so that modules are allocated to processors, and threads assigned within those independent modules, in a way that takes maximum advantage of parallel processor resources.

8. REFERENCES

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

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

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

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

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

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