PERFORMANCE PREDICTION AND CALIBRATION FOR A CLASS OF MULTIPROCESSOR SYSTEMS

D. Vrsalovic
D. Siewiorek
Z. Segall
E. Gehringer

22 August 1984

This research has been supported in part by the Ballistic Missile Defense Advanced Technological Center under contract DASG-60-80-C-0057, and by National Science Foundation grant MCS-8120270. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of BMDATC, NSF, Carnegie-Mellon University or the U.S. government.

Table of Contents

List of Figures

List of Tables

Abstract

We present a model for predicting multiprocessor performance on iterative algorithms which are made up of repeated *application cycles*. Each application cycle consists of some amount of access to global data and some amount of local processing. The application cycles may be synchronous or asynchronous, and the processors may or may not incur waiting time, depending on the relationship between the access time and processing time. We study how performance is affected by the processor and memory speed and the connection throughput. We also show how the model can be generalized to allow for processing and accessing to be interleaved throughout an application cycle. Significantly, the model permits the influence of the decomposition strategy upon the speedup to be separated from the influence of resource utilization. We study the decompositions of several sample algorithms, and identify several decomposition groups. The predictions of the model are compared with experimental results for three algorithms run on the Cm* multiprocessor. Finally, using the Poisson partial-differential equation algorithm as an example, we investigate how its decomposition affects its performance.

1 Introduction

With the decreasing cost of processors, it will soon be economically feasible to connect many of them together in a single tightly coupled multiprocessor system. Experimental systems, called "multiprocessor testbeds," have been built to investigate performance, but a comprehensive approach to performance prediction requires accurate models as well. In the experiment-preparation phase, reliable performance prediction reduces the experimentation space, thus reducing the experimentation time significantly. Once experimentation results are obtained the same models can be used to identify target system behavior and prepare new experiments, thus closing the experimentation cycle. Experimental results, in turn, aid in tuning the model. A model which initially provides only rough estimates can be refined to produce accurate results for a broad range of architectural parameters.

Most of the published multiprocessor models[1, 2, 3, 4] have been based on statistical methods, predicting statistical mean values for performance over some time interval. It was shown[5] that, in a real-time system, system performance in short-term *application cycles*—as well as long-term average behavior—is of interest. In order to evaluate this performance, the notion of *cyclic processing power* was defined as the effective number of processors (that is, the number of processors not idle due to contention) working cooperatively in every cycle.

Cyclic processing power is a measure of the system's processor utilization. Naturally one can assume that better utilization of processors will lead to better performance but, since there are several different parallel decompositions for the same algorithm, this is not necessarily true. Sometimes decomposition strategies which assign shorter sub-process iteration times also tend to lower the processing-to-global-memory access ratios of the parallel processes. Lower processing-to-access ratios decrease the cyclic processing power *CP*, which in turn lowers the speedup *SP*. It seems that in most cases, decomposition strategies and processing-to-access ratios influence performance in opposite directions. In this report, we will analyze the influence of the decomposition strategy as well as the influence of the processing-to-access ratio.

The expression given below for cyclic processing power[5] *CP* is based on a trivial multiprocessor model: a bus-oriented architecture with one common bus and a common memory, executing an infinitely decomposable parallel workload. The workload is assumed to have exactly one atomic (indivisible) access and one atomic processing period in every identical iteration, which we will call an *application cycle*, or simply an "iteration." Due to the large number of iterations initial start-up transients were neglected.

$$CP = \sum_{N} \frac{t_a + t_p}{t_a + t_p + t_w} \tag{1}$$

where

- t_p is the processing time within a cycle,
- t_a is the access time within a cycle, and
- t_w is the waiting time due to contention for common resources.

Assuming a balanced load, the sum in (1) degenerates to multiplication by N, resulting in the following expressions[5] for a synchronous (synchronization points at the beginning and end of each iteration), balanced load

$$CP = \frac{N}{1 + (N-1)\rho} \tag{2}$$

and an asynchronous balanced load (without any global synchronization points)

$$CP = \min[N, 1/\rho], \tag{3}$$

where ρ is defined as

$$\rho = \frac{t_a}{t_a + t_p} \tag{4}$$

The main disadvantage of this model is that it addresses a very small group of simple applications and multiprocessor architectures. A more refined model capable of describing a wider range of architectures and applications will be described in the following sections.

2 Refined Analytical Model

In order to develop a new model capable of dealing with a broader range of applications and systems, one should first address the main two weaknesses of the basic model. Such a new model should be capable of accurately describing a variety of network interconnection architectures, and of accounting for potential pipelining, caching, or distributed common resources.

Producing a model that is capable of describing the whole spectrum of hardware architectures and parallel workloads is impractical due to the large number of parameters such a model would require. The approach we take in this paper by modeling upper and lower performance bounds is more realistic, and can provide answers to two important design questions:

- What is the worst overhead that can be incurred by a parallel decomposition of some algorithm?
- How much can this performance be improved by changing the architecture or the implementation details, and is the improvement worth the price?

2.1 Architectural refinements

The general architecture of a multiprocessor system as considered in this text is presented in Figure 1.

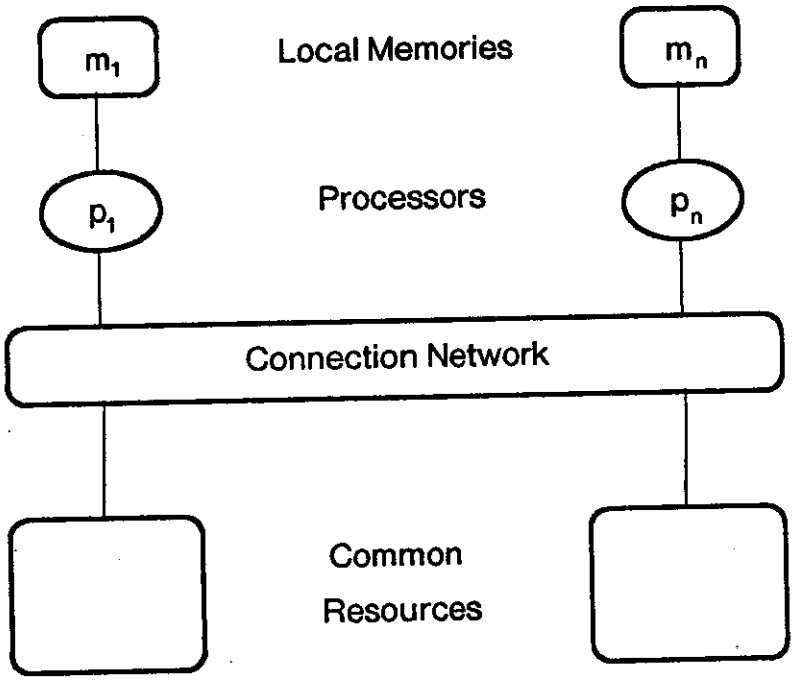

Figure 1: General architecture of a multiprocessor system

The system encompasses *N* processors, each having a considerable amount of a private memory for

local code and data. There is also a set of a common resources for storing global data. Processors and common resources communicate via a network. The connection network itself provides only circuit switching, not message switching. Requests for common resources are granted on a FIFO basis rather than on the basis of a priority scheme. These assumptions allow the waiting time t_w to be modeled as a linear function of N.

In addition to N, there are three other architectural parameters in the model:

- *ps*: processor speed relative to the speed of a *reference* processor, which we assign a speed of 1.
- *cas*: common access speed. The speed of accessing common memory relative to the speed of a *reference* memory with a speed of 1.
- *cat*: common access throughput. The value of this parameter is the number of processors which can simultaneously access a common resource without contention. This value depends not only upon hardware features (e.g. architecture of the interconnection network, and existence of multiple common resources), but also upon the algorithm decomposition and the partitioning of global data among multiple common resources.

A list of all model parameters is presented in Table 1. To distinguish the parameters of the uniprocessor solution from those of the decomposed processes, the former will be denoted by capitals and the latter by lower-case letters. The *processing-to-access ratio* of a workload is defined as—

$$X = \frac{T_p}{T_a}$$
$$x = \frac{t_p}{t_a} \qquad (5)$$

For the moment, let us assume that x is constant and independent of the number of processes in the parallel decomposition (i.e. $x = X$ for all N), although it will be seen later that this represents only a special case in practice. Further substitution of (5) in (2) and (3) gives—

$$CP = \frac{N(1+X)}{N+X} \qquad (6)$$

and

$$CP = \min[N, 1+X] \qquad (7)$$

Note a few intuitive interpretations of these formulas: When $N = 1$, $CP = 1$ for both the synchronous and asynchronous case. As X gets smaller, both the synchronous and asynchronous CP approach 1, indicating that an access-bound computation will show little speedup. For $N > 1$ and $X > 0$ (the usual

Table 1: Parameters of the Model

cas	*Common access speed*, the speed at which global memory can be accessed, relative to the speed of a *reference* memory. (page 5)
cat	*Common access throughput*, the number of processors which can simultaneously access a common resource without contention. (page 5)
CP	*Cyclic processing power*, the number effective number of processors working cooperatively in every cycle. (page 3)
f_a	The *access decomposition function*, the ratio of global access time in a uniprocessor implementation to global access time by an individual process in a multiprocessor implementation. (page 9)
f_p	The *processing decomposition function*, the ratio of processing time in a uniprocessor implementation to processing time for an individual process in a multiprocessor implementation. (page 9)
$I_{i=j,k}$	Notation for the serial iterations for i from j to k by 1. (page 18)
N	The number of processors in the system. (page 4)
ps	Processor speed relative to the *reference* processor. (page 5)
SP	*Speedup*, the ratio of the reference iteration time in a uniprocessor implementation to the iteration time of an individual process in a parallel implementation. (page 12)
SP^*	*Normalized speedup*, the ratio between the cycle time of a particular algorithm/implementation and that of the uniprocessor *basis solution*. (page 13)
t_a	Access time for a subprocess within a cycle. (page 3)
t'_a	*Modified access time*, the time it takes to make t_a accesses to a memory of speed cas. (page 7)
T_a	Access time for a uniprocessor implementation within a cycle. (page 5)
t_c	*Cycle time*, the sum of the iteration time ($t_p + t_a$) and the waiting time t_w. (page 12)
t_i	*Iteration time*, the sum of processing and access time for a single process in an iteration. (page 12)
T_i	*Iteration time* for the uniprocessor implementation. (page 12)
t_p	Processing time for a subprocess within a cycle. (page 3)
t'_p	*Modified processing time*, the time it takes a processor of speed ps to perform t_p units of work. (page 7)
T_p	Processing time for a uniprocessor implementation within a cycle. (page 5)
t_w	Waiting time due to contention for a subprocess within a cycle. (pages 3, 8)
T_1	Processing time for an iteration when local data copies are not used. (page 19)
T_2	Time to perform a single global access, assuming no contention. (page 19)
T_3	*Processing* time (for address calculation, etc.) associated with copying a single global data item to or from global memory. (page 19)

Table 1, cont.: Parameters of the Model

x	*processing-to-access ratio* for a subprocess, equal to t_p/t_a. (page 5)
x'	*Modified processing-to-access ratio*, defined as t'_p/t'_a. (page 7)
X	*Processing-to-access ratio* for a uniprocessor implementation, equal to T_p/T_a. (page 5)
δ	Average utilization of an individual processor with a *ps* of 1. (page 12)
$\mu_{cas/cat}$	*cas/cat-efficiency*, the (theoretical) percentage decrease in cycle time which would occur if *cas* and *cat* were infinite. (page 15)
μ_{ps}	*ps-efficiency*, the (theoretical) percentage decrease in cycle time which would occur if *ps* were infinite. (page 15)
ρ	The fraction of time a processor spends performing global accesses, $t_a/(t_a + t_p)$. (page 3)

case), the *CP* of an asychronous computation is always greater than for a sychrounous computation. We will now proceed to consider the influence of *ps*, *cas*, and *cat* on *CP*.

2.2 Influence of the processor and access speed

The only variable in equations (6) and (7) which is dependent on *ps* and *cas* is x. Increasing the processor speed or will shorten t_p. Similarly, increasing the common access speed will shorten t_a, and thus change x proportionately. The modified processing-to-access ratio x' will be—

$$t'_p = \frac{t_p}{ps}$$

$$t'_a = \frac{t_a}{cas}$$

and, consequently

$$x' = \frac{t'_p}{t'_a} = x\,\frac{cas}{ps} \tag{8}$$

Since we have assumed that $x = X$ for all processes, the substitution of x' for x in (6) and (7) gives—

$$CP = \frac{N(ps + cas\,x)}{(N\,ps + cas\,x)} \tag{9}$$

and

$$CP = \min[N,\ 1 + cas\,x/ps]$$

2.3 Influence of the connection throughput

To see what happens in a multiprocessor whose interconnection network allows more than one processor parallel access to global data without degradation, the original definition of *CP* (1) should be analyzed. Assuming a balanced load, (1) degenerates to—

$$CP = N\frac{t_p + t_a}{t_p + t_a + t_w} \tag{10}$$

The waiting time of a process is obviously depends on *cat*, the number of processes which can simultaneously access the data without contention. If parallel access is possible, *cat* measures this parallelism, and the waiting time for a process is proportional to the integer $\lfloor N/cat \rfloor$. (If there are seven processes, for example, and *cat* is two, then the seventh process must wait for three sets of two processes to complete their accesses; hence the waiting time is proportional to 3.) In order to keep the performance-prediction functions continuous, let us define the waiting time of a parallel process slightly differently,

$$t_w = \left[\frac{N}{cat} - 1\right] t_a \tag{11}$$

for the synchronous case, and

$$t_w = \max\left[0, \left[\frac{N}{cat} - 1\right] t_a - t_p\right] \tag{12}$$

for the asynchronous case.[5] Substituting the definitions of (11) and (12) and the result of (9) into the definition of *CP*, we derive—

$$CP = cat\, N\frac{ps + cas\, x}{N\, ps + cas\, cat\, x} \tag{13}$$

$$CP = \min[N, cat\,(1 + cas\, x/ps)], \tag{14}$$

which takes into account all three of the parameters defined above. It is interesting to predict the results of a "brute-force" application of a huge numbers of processors. A few conclusions can be drawn by analyzing the $\lim_{N\to\infty} CP(N)$:

- The maximal *CP* for synchronous systems is

  $$CP(\infty) = cat\,(1 + cas\, x/ps).$$

  - In the case of a large $cas\, x/ps$ (i.e. a large granularity of parallelism) a reasonable *CP* could be obtained using a conservative architecture with *cat* equal to 1 or nearly 1.

- Conversely, in a case of small *cas x/ps* (i.e. a small granularity of parallelism) the only means of improving *CP* is to use an architecture/algorithm combination with a larger *cat*.

- For an asynchronous system there is no gain at all by increasing *N* beyond

$$N = cat\,(1 + cas\,x/ps)$$

due to the fact that *CP* will not increase for *N* greater than this.

CP, however, is only one of the characteristics of a multiprocessor that affects performance, and a more detailed sensitivity analysis will be provided in Section 4, after parameters associated with the decomposition functions have been added to the model.

2.4 Decomposition of an algorithm into processes

When an algorithm is decomposed, local processing and common access are usually divided into *N* equal parts according to some rules. These rules can be expressed as functions of *N*, and need not be identical for local processing and global access. Consequently the processing-to-access ratio of a decomposed process will not be constant, but rather a function of *N*:

$$t_p(N) = \frac{T_p}{f_p(N)} \tag{15}$$

$$t_a(N) = \frac{T_a}{f_a(N)} \tag{16}$$

$$x(N) = \frac{t_p(N)}{t_a(N)} = \frac{T_p}{T_a}\,\frac{f_a}{f_p} = X\frac{f_a}{f_p} \tag{17}$$

The substitution of (17) into (13) and (14) gives the final expressions for *CP*:

- *synchronous case*—

$$CP = \frac{cat\,N(ps\,f_p + cas\,x\,f_a)}{N\,ps\,f_p + cas\,cat\,x\,f_a} \tag{18}$$

- *asynchronous case*—

$$CP = \min\,[N,\, cat\,(1 + cas\,X\,f_a/ps\,f_p)] \tag{19}$$

where f_p and f_a are the processing and access-decomposition functions respectively. Theoretically, a decomposition function can be any mathematical function of *N*, but as is shown in Section 5, some functions frequently appear in practice.

2.5 Variations in iteration cycle

Until now, all the expressions that we have seen for *CP* have been derived under the assumption that common access and local processing are atomic operations, and that each iteration has only one of each kind. In practice, this is seldom true—an iteration frequently consists of several interleaved processing and access periods. We shall now proceed to consider two special cases, representing the upper and the lower performance bounds of all possible implementations of an iteration:

- Lower-bound performance (the "worst case") is encountered when synchronization is required after each consecutive access-processing pair. The *CP* of a such system is—

$$CP = \sum_{N} \frac{\sum_{k} (t_{pk} + t_{ak})}{\sum_{k} (t_{pk} + t_{ak} + t_{wk})} \tag{20}$$

  where k is the number of processing-access pairs in the iteration. Due to the fact that there is a linear dependency between t_w and t_a and N, equation (20) yields the same result as would only one atomic (t_p, t_a) pair which had a duration equal to the length of an iteration.

- Upper-bound performance (the "best case") is achieved by the asynchronous case with only one atomic (t_p, t_a) pair whose processing and access times are equal to the sum of those which make up the iteration.

The worst case definition is intuitive, as there is obviously no worse case than for all of the N processes to request the common resources at exactly the same time, and then to wait for each other to finish in order to perform simultaneous accesses again. To establish that our "best case" is in fact optimal, however, requires a proof.

Claim: The asynchronous case of N parallel processes, each with only one atomic pair (t_p, t_a) per iteration yields the best performance of all cases with access and processing periods interleaved within the iteration.

Proof: For clarity and without loss of generality, we assume that *cat*, *cas*, and *ps* all have the value of unity. Given an iteration with a single (t_p, t_a) pair, suppose we rearrange it into M pairs (t_{pk}, t_{ak}) such that $\Sigma t_{pk} = t_p$ and $\Sigma t_{ak} = t_a$. For this case to be "better" than the original iteration then the total waiting time must be shorter than that of the first case. If the first case had no waiting time then it was optimal, and no further improvements are possible.

We now consider the case in which the original iteration has a positive waiting time. For this to happen, each process must finish its processing in less time than it takes for all the processes to complete their accesses. Hence $t_w = (N-1)t_a - t_p$ and the time to complete one iteration is $t_p + t_a + t_w$

$= Nt_a$. Since this is the time required for all processes to access the shared data, improvement on this time is not possible, so any optimal rearrangement must complete one iteration for all processes in Nt_a. Hence

$$\sum t_{ak} + \sum t_{pk} + \sum t_{wk} = Nt_a$$

$$t_a + t_p + \sum t_{wk} = Nt_a$$

$$\sum t_{wk} = (N-1)t_a - t_p = t_w$$

Therefore no rearrangement of the original one-pair iterations can reduce the total waiting time, so the one-pair asynchronous iterations are optimal as originally claimed.

3 Processing Power vs. Speedup

As we have seen in Section 1, cyclic processing power (*CP*) is a measure of the amount of computation which a system can perform in an algorithm-dependent *iteration time* t_i, which is defined as the sum of t_p and t_a. The *CP* of the normalized uniprocessor system is defined to be 1. In a multiprocessor system, δ, the average utilization of an individual processor is

$$\delta = CP/N \tag{21}$$

Intuitively, we expect the *CP* of an *N*-processor system to be higher than that of uniprocessor system, but not necessarily *N* times more powerful, since the processors sometimes have to wait for each other. The waiting time per cycle is t_w, and the *cycle time* is defined as the sum of the iteration time and the waiting time,

$$t_c = t_p + t_a + t_w. \tag{22}$$

Usually, the amount of waiting increases with the number of processors; this is reflected in a decreasing δ.

While it is obviously desirable for a system designer to keep resource utilization high, his main concern is to minimize the amount of time needed to complete an application. We will use *speedup factor* (*SP*) as a measure of the performance of a multiprocessor system compared with the reference uniprocessor system. Speedup has been defined in several different ways.[6] We shall define it as a ratio of cycle times:

$$SP(N) = \frac{T_c}{t_c(N)}$$

It is useful to express *SP* in a slightly different form to illustrate how it is influenced by the utilization δ and the decomposed iteration time t_i (note that $T_c = T_i$ due to the absence of degradation in the uniprocessor implementation):

$$SP(N) = \delta \frac{T_i}{t_i} \tag{23}$$

Speedup is thus the ratio between the reference iteration time and the decomposed iteration time, slowed by the utilization δ. In order to calculate speedup, taking into account the processor speed *ps* and the memory speed *cas*, we need the value of these quantities:

$$T_i = \frac{T_a}{cas} + \frac{T_p}{ps} \tag{24}$$

$$t_i = t_a + t_p = \frac{T_a}{cas\, f_a} + \frac{T_p}{ps\, f_p} \tag{25}$$

Substituting the results of (24) and (25) in (23) yields

$$SP(N) = \delta \frac{\frac{T_a}{cas} + \frac{T_p}{ps}}{\frac{T_a}{cas f_a} + \frac{T_p}{ps f_p}}$$

Using the definition of δ from equation (21) and substituting for CP using (18), we derive

$$SP = \frac{cat f_a f_p (ps + cas X)}{(N ps f_p + cas cat X f_a)} \tag{26}$$

for the synchronous case, and, substituting for CP using (19),

$$SP = \min\left[f_a f_p \frac{ps + cas X}{ps f_p + cas X f_a}, cat f_a \frac{ps + cas X}{N} ps \right] \tag{27}$$

From these equations, we can draw this general conclusion: While cyclic processing power depends only on the ratio of the decomposition functions f_p and f_a, speedup depends also on their values. In practical terms, this means that it is possible to decompose an algorithm into a set of processes which exhibit high processor utilization but low speedup.

Normalized Speedup

We have defined speedup (23) as a function of N, the number of processors in a system. As mentioned above, the implementor's real goal is to minimize execution time, not simply to maximize speedup. Toward this end, it may be necessary to change the implementation of an algorithm, the algorithm itself, or even the underlying system architecture to obtain better performance. Speedup, as defined above, is not an adequate measure of the improvement, because it cannot take into account any of these factors. Notice that all of these factors affect the decomposition functions f_p and f_a, as well as T_j. Therefore, let us define a more general term, *normalized speedup*, to measure the speed of a particular implementation with respect to a particular uniprocessor solution. We choose this *basis solution* to be the uniprocessor solution with all data global rather than local. Note that this solution is hypothetical, since we probably would not implement a uniprocessor program this way, but it does serve as a good basis for comparison.

Normalized speedup, then, is the ratio between the cycle time of a particular algorithm and implementation and that of the basis solution:

$$SP^* = SP \frac{T_{c_{basis}}}{T_c},$$

where T_c is the cycle time for the uniprocessor implementation of the same algorithm.

Now, let us define a *decomposition* as an ordered pair (f_p; f_a). Then the normalized speedup SP^* is the same as the relative speedup for an (N; N) decomposition.

4 Sensitivity of Performance to Architectural Changes

Until now, we have discussed only relative performance. It is, of course, very useful, especially in early design phases, to study how a proposed algorithm/architecture/implementation combination will perform relative to some hypothetical basis solution, but the designer's ultimate goal is always to predict absolute system performance. As mentioned first in Section 3, cycle time is the sum of iteration time and waiting time:

$$t_c = t_p + t_a + t_w$$

Both synchronous and asynchronous cases will be discussed using the following notation. The definitions are similar to those of Section 2.4, except that the expressions for t_p and t_a now take into account the processor speed *ps* and memory speed *cas*.

$$f_x = \frac{f_p}{f_a} \qquad X = \frac{T_p}{T_a}$$

$$t_a = \frac{T_a}{cas\, f_a} \qquad t_p = \frac{T_p}{ps\, f_p}$$

4.1 The synchronous case

Substitution of (11) into (22) gives

$$t_c = t_p + t_a + t_w = t_p + t_a + \left[\frac{N}{cat} - 1\right] t_a ,$$

or, using the above notation,

$$t_c = \frac{N T_a}{cas\, cat\, f_a} + \frac{T_p}{ps\, f_p} \tag{28}$$

Thus the cycle time is made up of two distinct components:

- A part proportional to the access time

$$t_{ca} = \frac{N T_a}{cas\, cat\, f_a},$$

- and a part proportional to the processing time (of one iteration).

$$t_{cp} = \frac{T_p}{ps\, f_p}$$

Although cycle time will always decrease as *ps*, *cas* or *cat* are increased, there will be some situations in which the t_{cp} or t_{ca} component will be so small that even bringing it down to near zero via huge investments in hardware would improve performance very little.

Let us define *ps-efficiency* μ_{ps}, and *cas/cat-efficiency* $\mu_{cas/cat}$, as the theoretical percentage decrease in cycle time when *ps*, or *cas* and *cat* are assumed to be infinite:

$$\mu_{ps} = \frac{t_{cp}}{t_{cp} + t_{ca}} = 1 \div \left[1 + \frac{N f_x ps}{X\, cas\, cat}\right] \times 100$$

$$\mu_{cas/cat} = \frac{t_{ca}}{t_{cp} + t_{ca}} = 1 \div \left[1 + \frac{X\, cas\, cat}{N f_x ps}\right] \times 100$$

It is probably more cost effective to concentrate on reducing the larger of μ_{ps} and $\mu_{cas/cat}$ by increasing *ps* or both *cas* and *cat*.

4.2 The asynchronous case

In the asynchronous case, there are two different possibilities:

- No waiting time:

$$t_c = t_p + t_a = \frac{T_p}{ps\, f_p} + \frac{T_a}{cas\, f_a}$$

The cycle time does not depend on *cat* and the efficiency factors are:

$$\mu_{ps} = 1 \div \left[1 + \frac{ps\, f_x}{cas\, X}\right] \times 100$$

$$\mu_{cas/cat} = 1 \div \left[1 + \frac{cas\, X}{ps\, f_x}\right] \times 100$$

- Waiting time equal to $[(N/cat) - 1]t_a - t_p$. This is the case where

$$\left[\frac{N}{cat} - 1\right] t_a > t_p$$

or

$$ps > \frac{X f_a cas}{f_p[(N/cat) - 1]} \qquad (29)$$

In this case, the cycle time does not depend on *ps*:

$$t_c = \frac{N}{cat} t_a$$

or, using the notation defined at the beginning of this section:

$$t_c = \frac{N T_a}{cas\, cat\, f_a} \qquad (30)$$

There is obviously no benefit to increasing *ps* beyond the value given by (29). The efficiency factors are $\mu_{ps} = 0$, and $\mu_{cas/cat} = 100$ for such an asynchronous case.

The design process inevitably involves tradeoffs between cost and performance. It is impossible to say which cost/performance tradeoffs should be made in a particular design, but the approach of balancing efficiency factors ($\mu_{ps} = \mu_{cas/cat}$) may be worthwhile to consider, at least in the initial design phases.

5 Decomposition Functions for Sample Parallel Applications

In this section, we will use straightforward implementations of several applications to illustrate a method for identification of decomposition functions. There is no loss of generality in considering simple applications, since the proposed methodology generalizes easily to more complicated applications.

In the following development, we will employ the following notation:

- Greek letters will be used to refer to global variables.
- The symbol $\mathcal{I}_{i=j,k}$ represents the serial iterations for i from j to k by 1.
- The symbol "$\leftarrow$" means assignment.
- The symbol "$\leftrightarrow$" indicates that values are copied in one direction and later copied back.

The algorithms described below may be implemented in several ways, but we will consider only two approaches, based on their use of shared data. If an algorithm requires multiple access to a shared data segment, we must choose whether or not each process is to maintain a local copy. We will present decomposition functions for both approaches in order to compare their performance.

Recall that in order to measure speedup, we always compare the performance of a particular decomposition to the performance of the reference solution, which is a uniprocessor implementation of the parallel algorithm. Strictly speaking, a uniprocessor implementation need not have any global data—all the data can simply be local to the single processor. For the sake of comparison, however, we will assume that the uniprocessor implementation uses exactly as much global data as its multiprocessor counterparts. In other words, to obtain the number of global references by the uniprocessor implementation, we derive an equation for the number of references by a multiprocessor implementation, and then simply set N to 1.

Regardless of whether data is local or global, it takes time to access it. Collectively, the time to perform the global data accesses makes up t_a. The time to perform the local accesses is included in t_p. When local copies are substituted for global data, t_p increases for two reasons: (1) it takes processing time to copy the data, and (2) the time for accessing the data is now charged to t_p instead of t_a. We will assume that the latter cost is negligible. In Section 6, we will consider the circumstances which make it worthwhile to make local copies of data.

5.1 Matrix multiplication

For $M \times M$ matrices A, B and C, the solution of the matrix equation

$$C = A \times B$$

can be decomposed into N processes,[7] each assigned to a submatrix of dimension $M/\sqrt{N} \times M/\sqrt{N}$. We can consider the submatrices themselves as arranged in m_1 rows and m_1 columns, and let c_{RS} be the Sth submatrix in the Rth row. In other words, this means that

$$m_1 = M/\sqrt{N}$$

$$(c_{RS})_{ij} = C_{[(R-1)m_1 + i,\ (S-1)m_1 + j]}$$

Now let $m_2 = \sqrt{N}$. If α and β refer to submatrices of A and B respectively, then the work done by an individual process is—

$$I_{i=1,m_1} I_{j=1,m_1} \{(c_{RS})_{ij} \leftarrow I_{Z=1,m_2} \Sigma_{k=1,m_1} (\alpha_{RZ})_{ik} \times (\beta_{ZS})_{kj}\} \tag{31}$$

If local copies are used, array values must be copied into and later out of the local workspace. The global accesses in (31) are replaced with local ones, and the following additional copying work must be done:

$$I_{Z=1,m_2} I_{i=1,m_1} I_{j=1,m_1} \{(a_{RZ})_{ij} \leftarrow (\mathrm{A}_{RZ})_{ij};\ (b_{ZR})_{ji} \leftarrow (\mathrm{B}_{ZR})_{ji}\} \tag{32}$$

Let

- T_1 be the processing time for one iteration without local copies (that is, the time for one iteration in (31))
- T_2 be the time to perform one global access, exclusive of waiting time, and
- T_3 be the processing time for copying a single global matrix element to or from global memory (that is, the time for one iteration in (32)).

Then we can derive the processing and access times (t_p and t_a) for an individual process by multiplying the number of iterations by the time needed for processing and accesses, respectively, within an iteration.

Without local copies:

$$t_p = (m_1 m_1 m_2 m_1) T_1 = \frac{M^3 T_1}{N}$$

$$t_a = (m_1 m_1 m_2 m_1) 2T_2 = \frac{2M^3 T_2}{N}$$

Hence,

$$T_p = M^3 T_1 \qquad T_a = 2M^3 T_2 \qquad X = \frac{T_1}{2T_2}$$

$$f_p = N \qquad f_a = N$$

With local copies:

$$t_p = \frac{M^3 T_1}{N} + \frac{M^2 T_3}{\sqrt{N}}$$

$$t_a = (\sqrt{N}\, m_1 m_1) 2T_2 = 2M^2 \frac{T_2}{\sqrt{N}}$$

$$T_p = M^3 T_1 + 2M^2 T_3 \qquad T_a = 2M^2 T_2 \qquad X = \frac{MT_1 + T_3}{2T_2}$$

$$f_p = \frac{(MT_1 + T_3)N}{MT_1 + T_3\sqrt{N}} \qquad f_a = \sqrt{N}$$

5.2 Poisson equation on a square grid

There are several algorithms for the parallel solution of a partial differential equations.[8, 9] Let us consider first the square-grid decomposition. Other decompositions will be discussed in Section 9.

A general definition for a Poisson equation is:

$$\Delta u = F \tag{33}$$

where Δ is the Laplace operator for n variables. For the solution in a plane ($n = 2$), equation (33) can be written in finite difference form:

$$f_{i+1,j} + f_{i-i,j} + f_{i,j+1} + f_{i,j-1} - 4f_{i,j} = F_{i,j}$$

Given a decomposition based on a square grid where every processor works on a square with dimensions $m_1 \times m_1$, m_1 is again equal to $M/\sqrt{N}$, and the action of an individual process can be described by—

$$I_{i=1,m_1} I_{j=1,m_1} (\varphi_{RS})_{ij} = (w-1)(\varphi_{RS})_{ij} + (w/4)[(\varphi_{RS})_{i-1,j} + (\varphi_{RS})_{i+1,j} + (\varphi_{RS})_{i,j-1} + (\varphi_{RS})_{i,j+1}] \tag{34}$$

where φ_{RS} is the submatrix defined by $(\varphi_{RS})_{ij} = \varphi_{(R-1)m_1 + i,(S-1)m_1 + j}$ and w is a weighting factor which controls the rate of convergence.

The global matrix "wraps around," so that the Mth row is adjacent to the 1st row, and the Mth column is adjacent to the first column. Hence the submatrices in the m_1st row are adjacent to the submatrices in the 1st row and those in the m_1st column are adjacent to those in the 1st column. Thus all the elements on a square boundary are shared with neighboring squares:

$$(\varphi_{RS})_{m_1+1,j} = (\varphi_{R,S+1})_{1,j} \text{ (etc., for all boundaries)}$$

If local copies of global data are made, the non-boundary elements need be copied only once at the

beginning of the process and copied back only once at the end. In the interim, they are accessed and updated by only a single process. We may assume that this cost is negligible compared with the cost of the entire algorithm. The significant work of copying is that required to copy all of the boundary elements on every iteration. This can be done by a single loop which copies elements on all four boundaries:

$$
\begin{aligned}
I_{k=1,m_1-1}\{ & (f_{RS})_{k,1} \leftrightarrow (\varphi_{RS})_{k,1}; && \text{left boundary} \\
& (f_{RS})_{1,k+1} \leftrightarrow (\varphi_{RS})_{1,k+1}; && \text{top boundary} \\
& (f_{RS})_{k+1,m_1} \leftrightarrow (\varphi_{RS})_{k+1,m_1}; && \text{right boundary} \\
& (f_{RS})_{m_1,k} \leftrightarrow (\varphi_{RS})_{m_1,k}\} && \text{bottom boundary}
\end{aligned} \tag{35}
$$

We can now use the information of equations (34) and (35) to derive the decomposition functions.

Without local copies:

Only the boundary elements need to be maintained in global shared memory; all other elements may be kept in non-shared memory (which may be "local" to the process which uses them). Different iterations of the nested loop in (34) will require different numbers of global accesses depending on how close the elements involved in the calculation are to a boundary. The integers in Figure 2 show the number of atomic global accesses required to update each element of a submatrix with $m_1 = 6$. For example, the 6's for the corner elements mean that each of its four neighbors must be read from global memory, and the element itself must both be read from and written back to global memory.

Each iteration of the loop in (34) can be assigned to a class based on the number of global accesses it makes. There are five different classes: 0, 1, 2, 5, and 6. The processing time for any iteration is $t_p = T_1$ regardless of the number of local accesses it makes, since a local access is assumed to take negligible time. The access time for an iteration depends on its class, with t_{ai}, the access time for a class-k iteration, equal to kT_2. Let n_k denote the cardinality of class k for the iterations on an individual submatrix.

Class 0: Submatrix iterations $[i,j]$ where $i = 3, \ldots, m_1 - 2$ and $j = 3, \ldots, m_1 - 2$.

$$t_{a0} = 0 \qquad n_0 = (m_1 - 4)^2$$

Class 1: Submatrix iterations $[2,j]$ and $[m_1 - 1, j]$ where $j = 3, \ldots, m_1 - 2$,
$[i,2]$ and $[i, m_1 - 1]$ where $i = 3, m_1 - 2$.

$$t_{a1} = T_2 \qquad n_1 = 4(m_1 - 4)$$

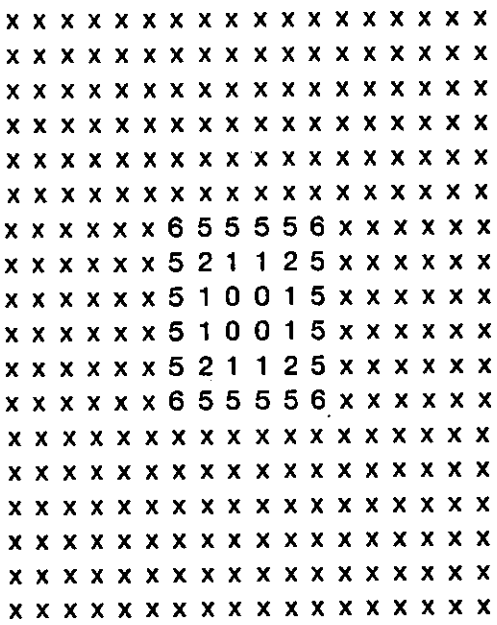

Figure 2: Square decomposition of a Poisson equation

Class 2: Submatrix iterations $[2,2]$, $[2,m_1-1]$, $[m_1-1,2]$, $[m_1-1,m_1-1]$

$$t_{a2}=2T_2 \qquad n_2=4$$

Class 5: Submatrix iterations $[1,j]$ and $[m_1,j]$ where $j=2,\ldots,m_1-1$,
$[i,1]$ and $[i,m_1]$ where $i=2,\ldots,m_1-1$.

$$t_{a5}=5T_2 \qquad n_5=4(m_1-2)$$

Class 6: Submatrix iterations $[1,1]$, $[1,m_1]$, $[m_1,1]$, $[m_1,m_1]$

$$t_{a6}=6T_2 \qquad n_6=4$$

From these expressions, it is easy to derive—

$$t_p = (m_1 m_1)T_1 = \frac{M^2 T_1}{N}$$

$$t_a = (6n_6 + 5n_5 + 2n_2 + n_1)T_2 = 24(m_1-1)T_2 = \frac{24MT_2}{\sqrt{N}} - 24T_2$$

$$= 24(M-1)T_2\left[\frac{M-\sqrt{N}}{(M-1)\sqrt{N}}\right]$$

$$T_p = M^2 T_1 \qquad T_a = 24T_2(M-1)$$

$$X = \frac{M^2 T_1}{24 T_2 (M-1)}$$

$$f_p = N \qquad f_a = \frac{(M-1)\sqrt{N}}{M-\sqrt{N}}$$

With local copies:

Two atomic accesses will be performed for each boundary element, one to read it and another to copy back its value at the end of the iteration, a total of $2\times4(m_1 - 1)$ atomic accesses per process. In order to update these boundary elements, an additional $4m_1$ accesses must also be performed to read boundary elements from neighboring processes. Each process, then, makes a total of $12m_1 - 8$ atomic accesses.

$$t_p = \frac{M^2 T_1}{N} + \frac{12MT_3}{\sqrt{N}} - 8T_3 = T_p \frac{M^2 T_1 + 12M\sqrt{N}\,T_3 - 8NT_3}{T_p N}$$

$$t_a = \frac{12MT_2}{\sqrt{N}} - 8T_2 = (12M-8)T_2\left[\frac{12M}{(12M-8)\sqrt{N}} - \frac{8}{(12M-8)}\right]$$

$$T_p = M^2 T_1 + 12MT_3 - 8T_3 \qquad T_a = (12M-8)T_2 \qquad X = \frac{M^2 T_1 + 12MT_3 - 8T_3}{(12M-8)T_2}$$

$$f_p = \frac{N(M^2 T_1 + 12MT_3 - 8T_3)}{M^2 T_1 + 12MT_3\sqrt{N} - 8NT_3}$$

$$f_a = \frac{(12M-8)\sqrt{N}}{12M - 8\sqrt{N}}$$

5.3 Linear differential equations

A linear differential equation of the nth order can be transformed into a set of n linear differential equations of the first order:

$$O^{n+1} = A \times O^n + B \times i^n$$

where A and B are matrices, O is a state vector, and i is an input vector. If each processor is assigned to a set of M/N adjacent matrix rows, the work of the Rth parallel process can be described as follows, letting Ω refer to the potentially global realization of O and ι to the potentially global version of i:

$$I_{i=1,\,M/N}\{(\Omega_R)_i^{(n+1)} \leftarrow I_{k=1,\,M}[A_{ik} \times (\Omega)_k^{(n)} + B_{ik} \times \mathrm{I}_k^{(n)}]\} \tag{36}$$

If local copies are to be made, the jth process must copy I and Ω at the beginning of each iteration, and afterwards, copy back the updated values from O_j. Hence, its additional work is—

$$I_{j=1,\,M}\{O_j \leftarrow \Omega_j;\ i_j \leftarrow \mathrm{I}_j\} \tag{37}$$

It is easy to derive the decomposition functions for one of the component processes described by (36):

Without local copies:

$$t_p = \frac{M^2 T_1}{N}$$

$$t_a = M(2M+1)T_2 \frac{1}{N}$$

$$T_p = M^2 T_1 \qquad T_a = M(2M+1)T_2 \qquad X = \frac{MT_1}{(2M+1)T_2}$$

$$f_p = N \qquad f_a = N$$

With local data copies:

$$t_p = \frac{M^2 T_1}{N} + 2MT_3 = (M^2 T_1 + 2MT_3)\frac{MT_1 + 2T_3 N}{(MT_1 + 2T_3)N}$$

$$t_a = 2MT_2 + \frac{MT_2}{N} = 3MT_2\left[\frac{1}{3N} + \frac{2}{3}\right]$$

$$T_p = M^2 T_1 + 2MT_3 \qquad T_a = 3MT_2 \qquad X = \frac{MT_1 + 2T_3}{3T_2}$$

$$f_p = \frac{(MT_1 + 2T_3)N}{MT_1 + 2NT_3}$$

$$f_a = \frac{3N}{2N+1}$$

5.4 Fourier transform

Given a complex sequence u_j for $j = 1, \ldots, M$, the discrete complex periodic Fourier transform is given by

$$U_k = (1/M)\sum_{j=1}^{M} u_j e^{-ikj2\pi/M}$$

and has the inverse

$$u_j = \sum_{k=1}^{M} U_k e^{ikj2\pi/M}$$

Thus, on each iteration it is necessary to produce M new values. One reasonable decomposition is to parcel out these U_k or u_j evenly among the processors. Given such a decomposition, the work of an individual process can be described as—

$$I_{r=1,M/N}\,(u_R)_r \leftarrow \sum_{k=1}^{M} \Upsilon_k e^{ik[(R-1)(M/N)+r]2\pi/M} \tag{38}$$

where Υ represents the potentially global realization of U. The additional work to make a local copy of Υ is

$$I_{j=1,M} U_j \leftarrow \Upsilon_j \tag{39}$$

Without local copies:

$$t_p = \frac{M^2 T_1}{N}$$

$$t_a = \frac{M^2 T_2}{N}$$

$$T_p = M^2 T_1 \qquad T_a = M^2 T_2 \qquad X = \frac{T_1}{T_2}$$

$$f_p = N$$

$$f_a = N$$

With local copies:

$$t_p = \frac{M^2 T_1}{N} + MT_3 = (M^2 T_1 + MT_3)\left[\frac{MT_1 + NT_3}{(MT_1 + T_3)N}\right]$$

$$t_a = MT_2$$

$$T_p = M^2 T_1 + MT_3 \qquad T_a = MT_2 \qquad X = \frac{MT_1 + T_3}{T_2}$$

$$f_p = \frac{(MT_1 + T_3)N}{MT_1 + NT_3}$$

$$f_a = 1$$

5.5 Parallel search

If an ordered vector $G = g_i, i = 1, \ldots, M$ is to be searched for a key,[10] a binary search would probably be used. A parallel decomposition could assign interleaved vector segments to several processors in order to speed up the search for a single key y. In the case that the maximum $\log(M/N)$ iterations are required to locate the key, the work can be described by—

$$I_{k=1,\log(M/N)}\{j \leftarrow \text{sign}[(\Gamma_R)_i - y];\ i \leftarrow i + j\left[\frac{\log(M/N)}{2^k}\right]\} \tag{40}$$

(Here Γ is the potentially global vector G.) There can be no performance improvements via using local copies since each global data element is accessed only once in (40), but since each processor is using a different segment of the global data, *cat* can easily be made larger than 1.

Parallel search for a single key:

$$t_p = \log(\frac{M}{N} + 1)T_1 = (\log M + 1)T_1\left[\frac{\log M - \log N + 1}{\log M + 1}\right]$$

$$t_a = \log(\frac{M}{N} + 1)T_2 = (\log M + 1)T_2\left[\frac{\log M - \log N + 1}{\log M + 1}\right]$$

$$T_p = (\log M + 1)T_1 \qquad T_a = (\log M + 1)T_2 \qquad X = \frac{T_1}{T_2}$$

$$f_p = \frac{\log M + 1}{\log M - \log N + 1}$$

$$f_a = \frac{\log M + 1}{\log M - \log N + 1}$$

Parallel search for multiple keys:

Given a vector of key values $Y = y_k, k = 1, \ldots, K$, a parallel decomposition could assign the y_k to processes according to the strategy—

$$I_{k=1,K/N} I_{r=1,\log M}\{j \leftarrow \text{sign}[\Gamma_i - (y_R)_k];\ i \leftarrow i + j \log \frac{M}{2^r}\} \tag{41}$$

If local copies are to be made the additional work would be:

$$I_{n=1,M}\{g_i \leftarrow \Gamma_i\} \tag{42}$$

From this, the following decomposition functions can be derived:

Without local copies:

$$t_p = \frac{K(\log(M) + 1)T_1}{N}$$

$$t_a = \frac{K(\log(M) + 1)T_2}{N}$$

$$T_p = K(\log(M) + 1)T_1 \qquad T_a = K(\log(M) + 1)T_2 \qquad X = \frac{T_1}{T_2}$$

$$f_p = N$$

$$f_a = N$$

With local copies:

$$t_p = \frac{K(\log(M)+1)T_1}{N} + MT_3 = T_p\left[\frac{K(\log M+1)T_1 + MNT_3}{(K(\log M+1)T_1 + MT_3)N}\right]$$

$$t_a = MT_2$$

$$T_p = K(\log(M)+1)T_1 + MT_3 \qquad T_a = MT_2 \qquad X = \frac{K(\log(M)+1)T_1 + MT_3}{MT_2}$$

$$f_p = \frac{(K(\log M+1)T_1 + MT_3)N}{K(\log M+1)T_1 + MNT_3}$$

$$f_a = 1$$

6 Single global data versus local global data copies

Since parallel processes often use the same global data items more than once during an iteration, it is a viable design alternative to make a local copy of the data so that only the first reference to each item needs to access a common resource. This section will investigate the local/global data tradeoff.

It was shown in Section 5 that changes in global data allocation and management induce changes in the processing-to-access ratio X and the decomposition functions f_p and f_a.

Claim: For any combination of decomposition functions of the form

$$f_p = N^j \qquad f_a = N^k \tag{43}$$

the lower-bound speedup curve will have an extreme *unless* $j = k + 1$ or $k = 1$.

Proof: Suppose that there exists a pair of decomposition functions for which SP has no extremes. Then the first derivative of SP (from equation (26)) should be independent of N. Letting ps, cas, and $cat = 1$ in equation (26), we derive this expression for SP:

$$SP = \frac{(1 + X) f_p f_a}{N f_p + X f_a}$$

We can set the derivative of SP equal to zero—

$$\frac{dSP}{dN} = 0$$

and solve for the value of N at which the maximum speedup is obtained. Substituting the values from (43), we want to solve

$$\frac{d}{dN} \frac{(1 + X) f_p f_a}{N f_p + X f_a} = 0$$

the numerator of which simplifies to

$$N^{j-k+1}(k - 1) + jX = 0$$

There will be a solution to this equation unless $k = 1$ or $k = j + 1$, in which case the speedup curve has no extreme.

Among the decompositions which have no maximum speedup are those of the form $(N^j; N)$. Of these, the $(N; N)$ decomposition is of particular interest since it is characteristic of several of the non-copying algorithms from Section 5. In fact, that is one reason why normalized speedup (Section 3) was defined in terms of an $(N; N)$ decomposition.

For decompositions which have a maximum speedup at N_{max} processors, increasing the number of processors past N_{max} will actually degrade performance. If enough processors are added, the

decomposition will eventually be outperformed by the $(N;N)$. Let us call this crossover point N_c. It is the smallest number of processors for which the $(N;N)$ decomposition performs better.

It is very difficult to give an expression for N_c in the general case, but considering the special case of matrix multiplication, as described in Section 5.1, let us make the assumption that the time to iterate through a row is much greater than the processor time to copy one data element, i.e., that $MT_1 \gg T_3$. In the expressions below, the subscript 1 refers to the implementation without local copies, and the subscript 2 refers to the copying version. Then

$$f_{p_1} = f_{a_1} = N$$

$$f_{p_2} = \frac{(MT_1 + T_3)N}{MT_1 + T_3\sqrt{N}} \qquad f_{a_2} = \sqrt{N}$$

$$T_{p_1} = M^3T_1 \qquad T_{a_1} = 2M^3T_2 \qquad X_1 = \frac{T_1}{2T_2}$$

$$T_{p_2} = (MT_1 + T_3)M^2 \approx M^3T_1 \qquad T_{a_2} = 2M^2T_2 \qquad X_2 = \frac{MT_1 + T_3}{2T_2} \approx MX_1$$

Except for very small matrices, it is likely that the time to iterate through a row is much greater than the processor time to copy one data element. Assuming that

$$f_{p_2} \approx \frac{M(T_1/T_3)N}{M(T_1/T_3)\sqrt{N}} = \frac{MkN}{Mk + \sqrt{N}} \quad \text{where } k = \frac{T_1}{T_3}$$

We want to determine the number of processors N_c where the performance of an $(N;N)$ decomposition just equals that of the local-copy implementation. That is, we want to determine N_c such that $SP_1 = SP_2^*$. (Recall that SP^* refers to normalized speedup.) If we let *pt*, *cas*, and *cat* all equal 1 in (26), we obtain

$$SP = \frac{f_a f_p(1 + X)}{(Nf_p + Xf_a)}$$

Thus,

$$SP_2^* = \frac{f_{a_2} f_{p_2}(1 + X_2)}{(Nf_{p_2} + X_2 f_{a_2})} \cdot \frac{T_{c_{basis}}}{T_c} \tag{44}$$

where $T_c = T_a + T_p$ is the cycle time for the uniprocessor implementation of an algorithm, and "basis" refers to the non-copying implementation. We also have

$$SP_1 = \frac{(1 + X_1)N^2}{N^2 + X_1N} \tag{45}$$

Setting (44) and (45) equal, after some algebra, we arrive at

$$N_c - M\sqrt{N_c} + \frac{X_1}{k} = 0$$

which can be solved as a quadratic equation to yield

$$N_c = \left[\frac{M + \sqrt{M^2 - 2T_3/T_2}}{2}\right]^2$$

Assuming that $T_3 \ll T_2$, we obtain

$$N_c \approx M^2$$

In other words, if T_3 is insignificant with respect to T_2, it is profitable to make local copies as long as the number of processors is at least somewhat less than the number of matrix elements. However, as T_3 increases with respect to T_2, N_c tends to decrease.

On the other hand, for a decomposition like the local-copies version of matrix multiplication, the maximum speedup is attained with a much smaller number of processors. The value of N which yields maximum performance can be calculated from

$$\frac{dSP^*}{dN} = 0$$

For the case of the local-copies version of matrix multiplication, (44) becomes

$$SP_2^* = \frac{f_{a_2} f_{p_2} M(1 + X_1)}{(Nf_{p_2} + X_2 f_{a_2})} = \frac{kNM(1 + X_1)}{N\sqrt{N}\ k + MkX_1 + X_1\sqrt{N}}$$

To solve for the extreme, we set the numerator of the derivative dSP_2/dN equal to zero:

$$M(1 + X_1)k \cdot \left[N\sqrt{N}\ k + MkX_1 + X_1\sqrt{N} - N \cdot \frac{3}{2} \cdot \sqrt{N}\ k - N \cdot X_1 \frac{1}{2\sqrt{N}}\right] = 0$$

Assuming that $M^2 \gg X_1$, this is a cubic equation in $\sqrt{N}$ whose solution is $N = (2MX_1)^{2/3}$.

7 Some Common Decomposition Functions

In order to evaluate the theoretical results, an emulation system was built for Cm*,[11] a multiprocessor with a structure similar to the modeled system. It consists of a user-interface process and 25 identical test processes, each emulating an atomic (t_p, t_a) pair. Test processes communicate with the user interface via a global vector allocated to the processor on which the interface process is running. Each test process shares a different segment of this vector with the interface, where its control variables are stored. All test processes share another segment which represents shared data.

The control variables of a process are defined as the reference values of t_p and t_a, the number of iterations per experiment, and the presence or absence of interprocess synchronization. The user may change the value of a specific control variable and start the experiments by commands to the interface process.

Each test process records its starting and finishing times, and reports them to the interface at the end of each experiment. Unfortunately, time measurements are invasive, and contention for the system's single clock register perturbs the measurements.

Besides the invasive measurement technique, certain model imperfections and emulator characteristics perturb the measured behavior. Some of them are caused by the fact that the interconnection network in Cm* can only approximate a multiprocessor architecture with a circuit switching network. If it were not for these perturbations, the measured behavior would be exactly as predicted, since the emulated workload exactly matches the modeled system. These measurements, then, do not address the question of whether the emulated processes are similar to actual parallel applications.

Imperfections in the model and the emulator can be summarized as follows:

- Invasive experimentation environment (especially due to timing measurements).
- Effects related to modeling a processing time which is usually of the form

$$t_p = T_{p_{fixed}} + \frac{T_{p_{var}}}{f_p}$$

instead of the

$$t_p = \frac{T_p}{f_p}$$

assumed by the model. $T_{p_{fixed}}$ is partially due to the experimentation environment.

- Effects connected with modeling an access time which is usually of the form

$$t_a = T_{a_{fixed}} + \frac{T_{a_{var}}}{f_a}$$

instead of the

$$t_a = \frac{T_a}{f_a}$$

assumed by the model. $T_{a_{fixed}}$ is partially due to the monitoring, and atomic access generation.

- The influence of nonuniform atomic access times due to the differences between the intercluster and intracluster access times in Cm*.
- Delays from queueing and processing by the Kmaps rather than the pure circuit-switching assumed by the model.
- Effects of the software-locking routines used by the emulator to implement atomic global accesses of variable duration. As the number of processors increases, t_a grows shorter, and eventually the computation and access time required to set and release the lock tends to perturb the relationship between t_p and t_a.

Four characteristic (f_a, f_p) pairs were simulated for $X = 10$, $X = 35$, and various numbers of processors. In all the calculations and measurements *cat*, *cas*, and *ps* are assumed to be 1. In this section, emulation results along with predicted values are presented. For each of four major decomposition groups, a mathematical expression for the speedup based on equations (26) and (27) is given. In addition the theoretical maximum speedup is calculated from:

$$\frac{dSP}{dN} = 0 \qquad \text{for the synchronous case, and}$$

$$N ps f_p - cat(ps f_p + cas X f_a) = 0 \qquad \text{for the asynchronous case (from (19).}$$

7.1 (*N*; *N*) decomposition group

Synchronous case—Figure 3:

$$SP = \frac{(1 + X)N}{N + X}$$

$$N_{max} = \infty$$

$$SP_{max} = 1 + X$$

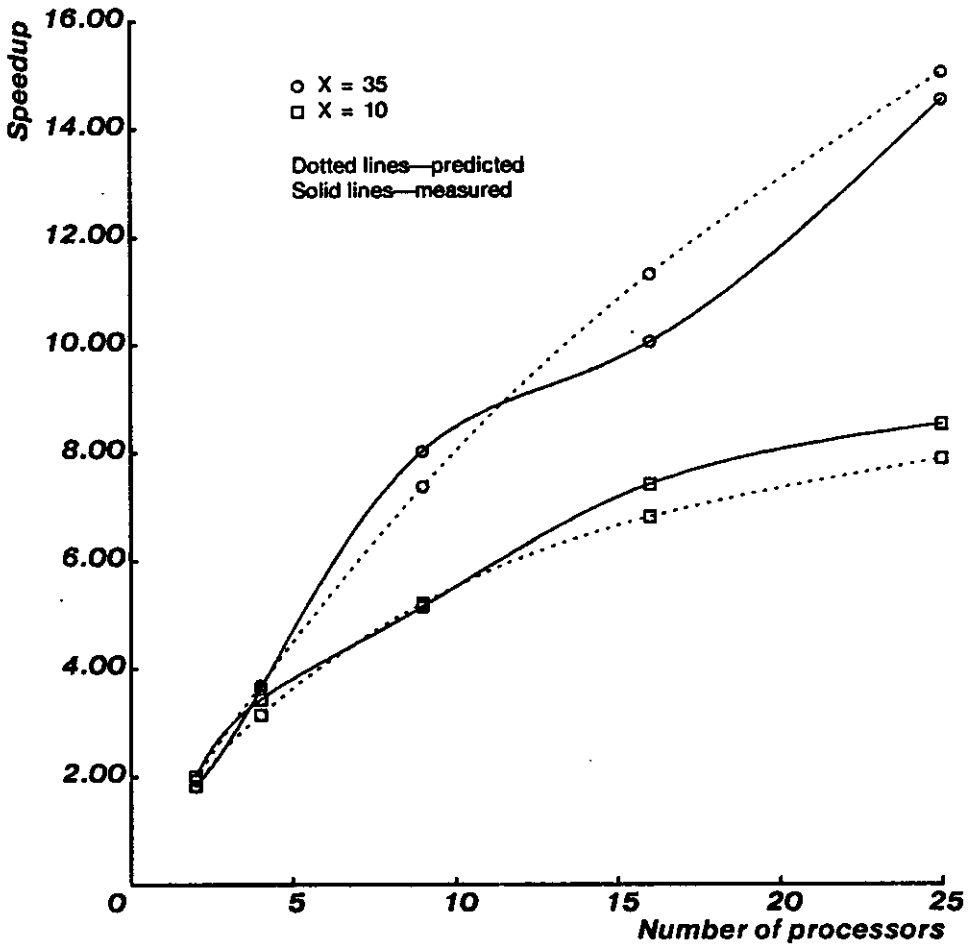

Figure 3: Speedup versus *N* for decomposition (*N*; *N*) and synchronous implementation

Asynchronous case—Figure 4:

$$SP = min[N, 1 + X]$$

$$N_{max} = 1 + X$$

$$SP_{max} = 1 + X$$

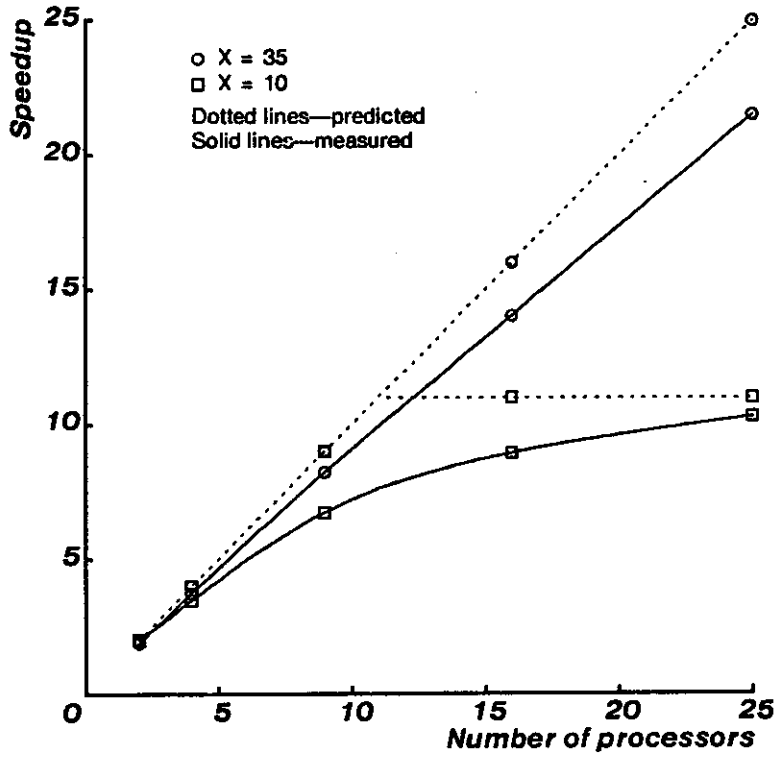

Figure 4: Speedup versus N for decomposition $(N; N)$ and asynchronous implementation

7.2 $(N; \sqrt{N})$ decomposition group

Synchronous case—Figure 5:

$$SP = \frac{(1 + X)N}{N^{3/2} + X}$$

$$N_{max} = (2X)^{2/3}$$

$$SP_{max} = \frac{2^{2/3}(1 + X)}{3X^{1/3}}$$

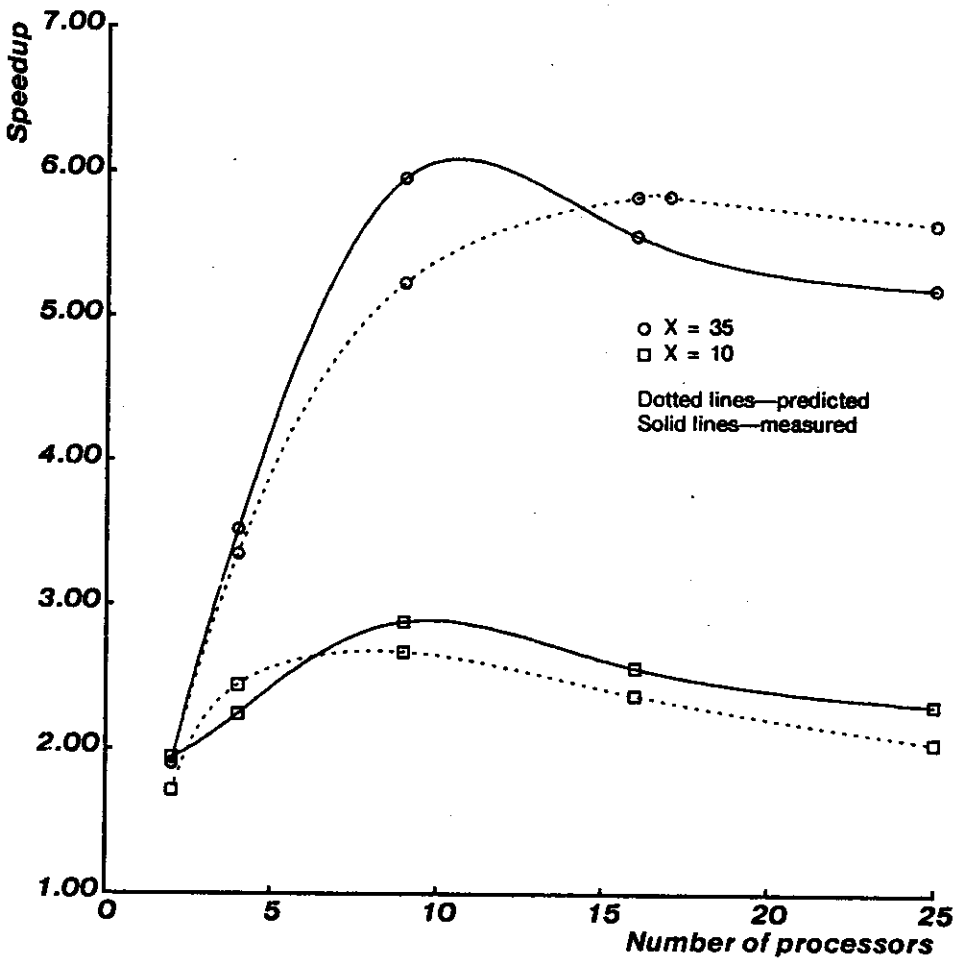

Figure 5: Speedup versus N for decomposition $(N; \sqrt{N})$ and synchronous implementation

Asynchronous case—Figure 6:

$$SP = \min\left[\frac{(1+X)N}{\sqrt{N}+X}, \frac{1+X}{\sqrt{N}}\right]$$

$$N_{max} = X^{2/3*}$$

$$SP_{max} = \frac{(1+X)X^{1/3}}{1+X^{2/3}}$$

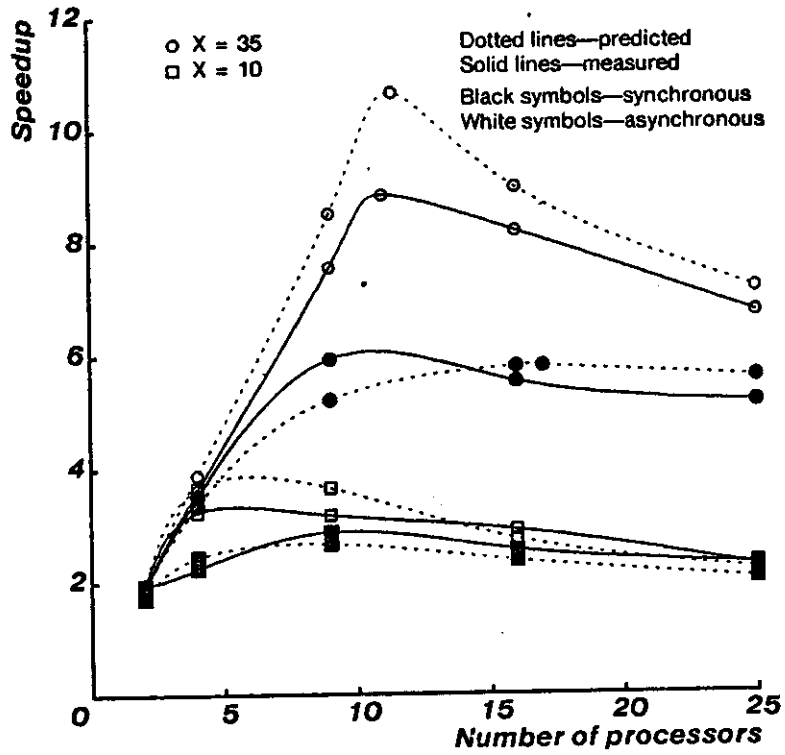

Figure 6: Speedup versus N for decomposition (N; $\sqrt{N}$) and asynchronous implementation

---

*approx. solution for $X^2/4 > 1/9$

7.3 (*N*; *1*) decomposition group

Synchronous case—Figure 7:

$$SP = \frac{(1+X)N}{N^2+X}$$

$$N_{max} = \sqrt{X}$$

$$SP_{max} = \frac{1+X}{2\sqrt{X}}$$

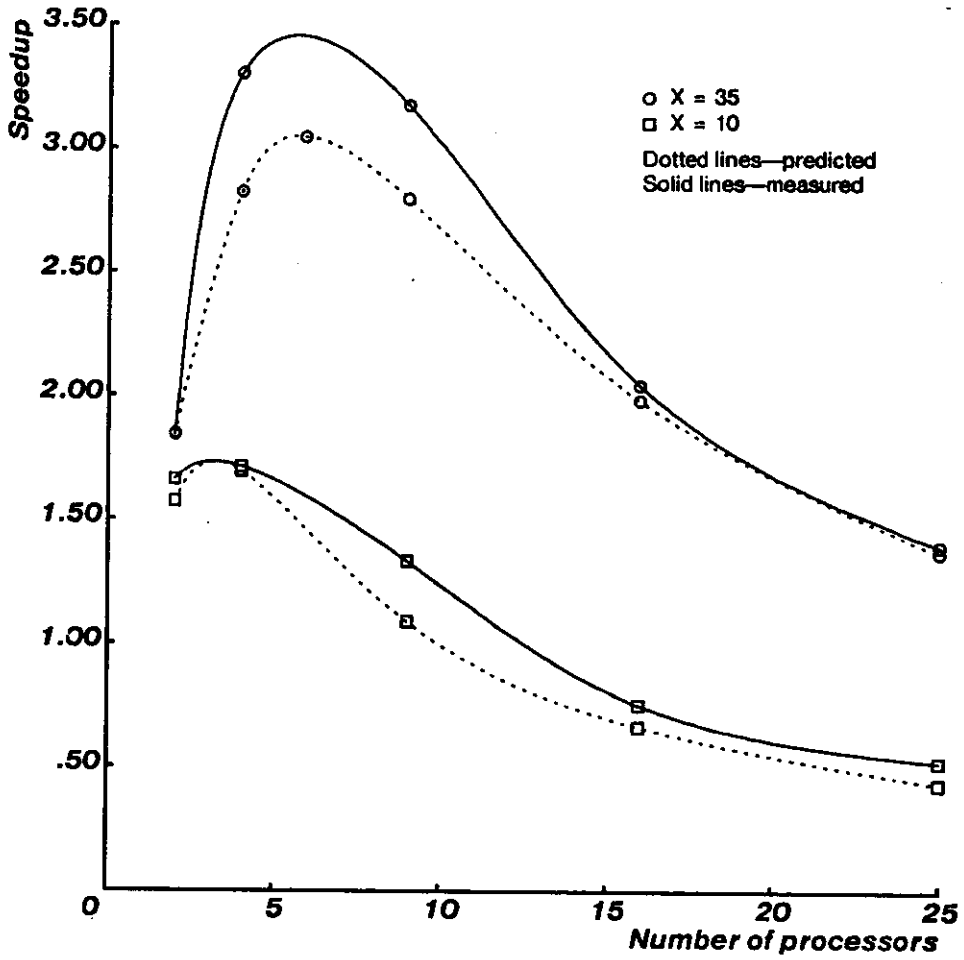

Figure 7: Speedup versus *N* for decomposition (*N*; 1) and synchronous implementation

Asynchronous case—Figure 8:

$$SP = \min\left[\frac{N(1+X)}{(N+X)}, \frac{1+X}{N}\right]$$

$$N_{max} = \frac{1+\sqrt{1+4X}}{2}$$

$$SP_{max} = \frac{2(1+X)}{1+\sqrt{1+4X}}$$

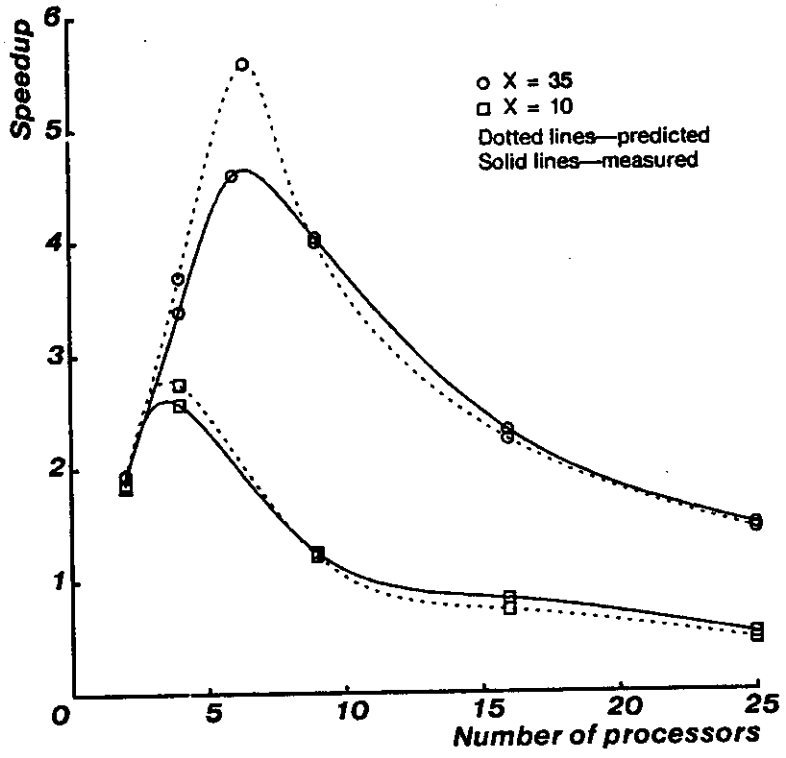

Figure 8: Speedup versus N for decomposition (N; 1) and asynchronous implementation

7.4 (log *N*; log *N*) decomposition group

Synchronous case—Figure 9:

$$SP = (1 + X)\frac{\log N}{N + X}$$

N_{max} can be derived from $N_{max}(\log N_{max} - 1) = X$**

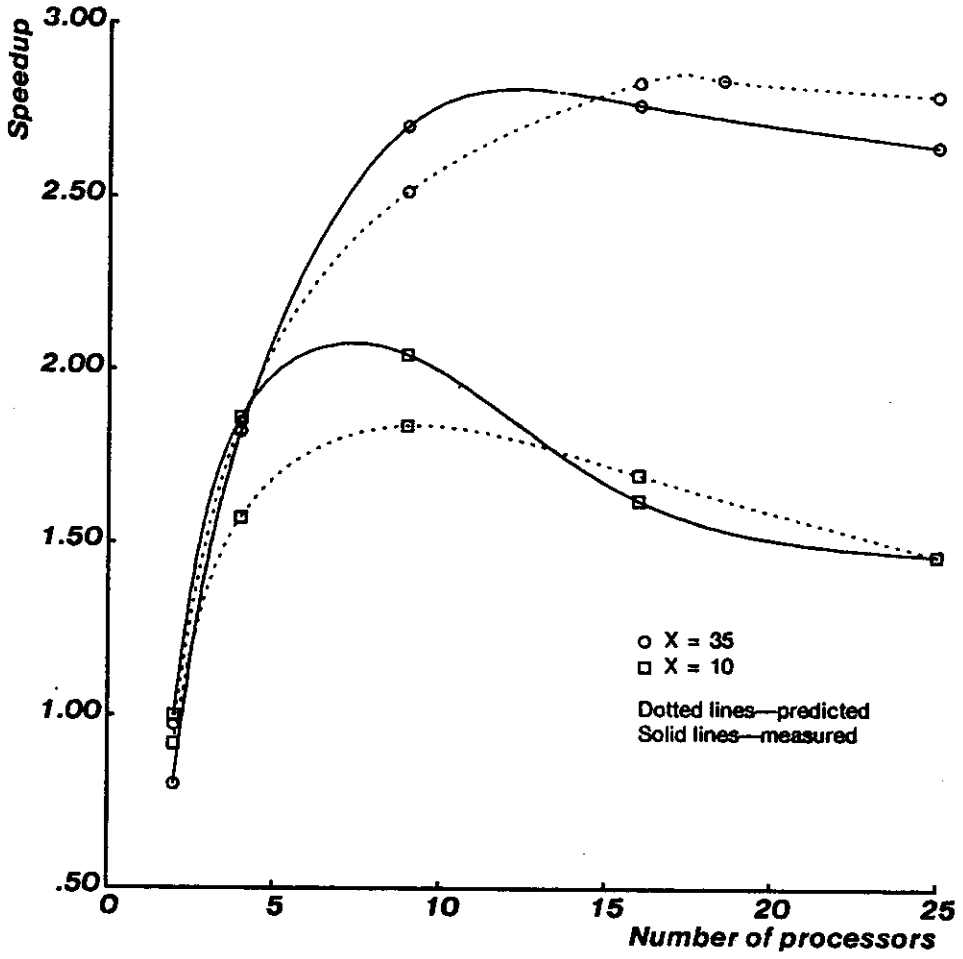

Figure 9: Speedup versus *N* for decomposition (log *N*; log *N*) and synchronous implementation

** numeric solution only

Asynchronous case—Figure 10:

$$SP = \min\left[\log N, \frac{(1+X)\log N}{N}\right]$$

$$N_{max} = 1 + X$$

$$SP_{max} = \log(1+X)$$

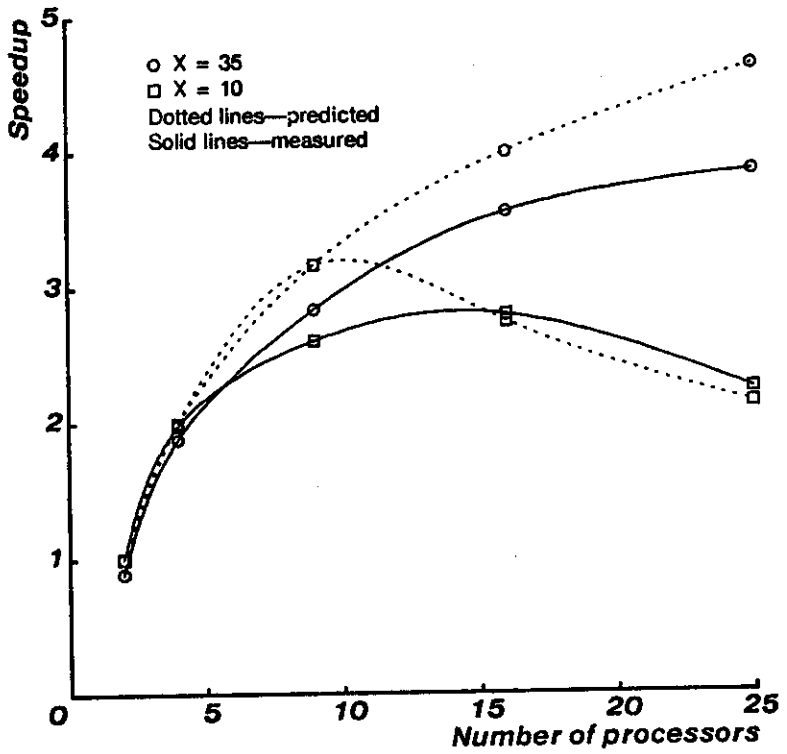

Figure 10: Speedup versus N for decomposition ($\log N$; $\log N$) and asynchronous implementation

7.5 Linear speedup

We now present one final decomposition group, which has not been tested empirically. This decomposition group exhibits *linear speedup*: a speedup of N with N processors. Although there are some exceptions,[12] linear speedup is the best that can be expected from most algorithms. An (N; N^2) decomposition induces linear speedup, since

$$\frac{N^2N(1+X)}{NN+XN^2} = N$$

It is interesting to note that this speedup is independent of X.

8 Correlation with Real Workload Implementations

In addition to performance prediction, another very important phase of the experimentation cycle is called *identification*. Identification is the process of describing the decomposition functions of a particular parallel workload mathematically, so that specific changes can be recommended to bring about performance improvement.***

In this section the results of three previously published experiments are used. Because the original source code was not readily available, it was necessary to estimate the relationship between T_1, T_2, and T_3. These values were estimated by attempting to fit the predicted curves to the measured curves, but it should be noted that the same choice of values produces a good fit for all the curves on each graph.

8.1 Processor speed variations

This experiment illustrates the influence of processor speed on performance. The parallel workload was an implementation on the Cm* multiprocessor of a molecular-dynamics algorithm. It consists of a number of parallel processes each calculating the binding energy between particles. After all calculations are done, the final result is saved as global data to be used in the next iteration. This description implies that local processing time for a decomposed process is variable and equal to—

$$t_p = \frac{T_p}{N}$$

The global access time is fixed and equal to the time required for returning a final result to global memory. Therefore

$$t_a = T_a$$

Consequently the decomposition functions are

$$f_p = N \qquad f_a = 1$$

To assure an atomic access for every process, a locking mechanism is used. Speedup was measured as a function of the processor speed. While the speed of the processor hardware could not be varied, faster processors could be simulated by replacing the slow LSI-11 floating-point calculations by "synthetic procedures" which took less time than the calculations and returned arbitrary results. (The goal was to study the effect of processor speed, so it did not matter that the results were incorrect.) Since only the floating-point calculations were "speeded up," the processor speed variation affected only one portion of T_p (and consequently only one portion of X).

*** Experimental results have been published in the referenced literature,[13, 7] and are reproduced here, courtesy of Robert A. Whiteside, for the purpose of comparison with the theoretical predictions.

Figure 11 shows both measured and theoretical results, assuming an estimated X. Although the synchronous case is supposed to be the "worst case," the measured performance for slow processors is worse than the theoretical lower bound due to the inability of this implementation to assure a balanced load. (Some processors have much more work to do than others.) Therefore when processors are slow, and t_p is dominant in the iteration, the idle processors induce a non-monotonic performance curve, and exhibit worse performance than theoretically predicted for a balanced load.

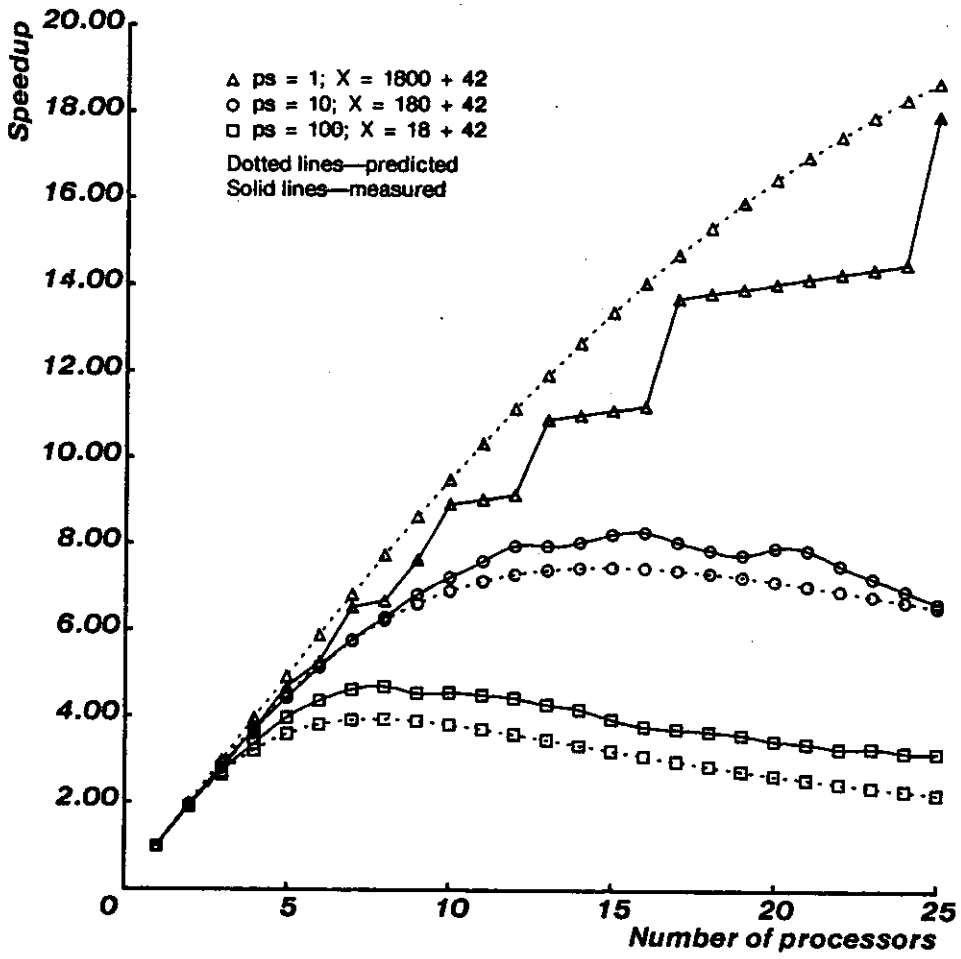

Figure 11: Molecular Dynamics Speedup, Dependency on Processor-Speed Variations

8.2 Speedup versus synchronization

In the second experiment, speedup was measured for various degrees of synchronization between processes. Experimental and theoretical results for the two extreme cases (full synchronization and no synchronization) are given in Figure 12. The X of the synchronous implementation is only about

half as large as the X of its asynchronous counterpart, due to the fact that the omitted synchronization code consists mainly of access to common data.

The implementation for molecular dynamic simulation analyzed here is a typical one-pair implementation (the kind considered in the proof of Section 2.5) and, for that reason, the measured results are very close to the upper and the lower bound respectively.

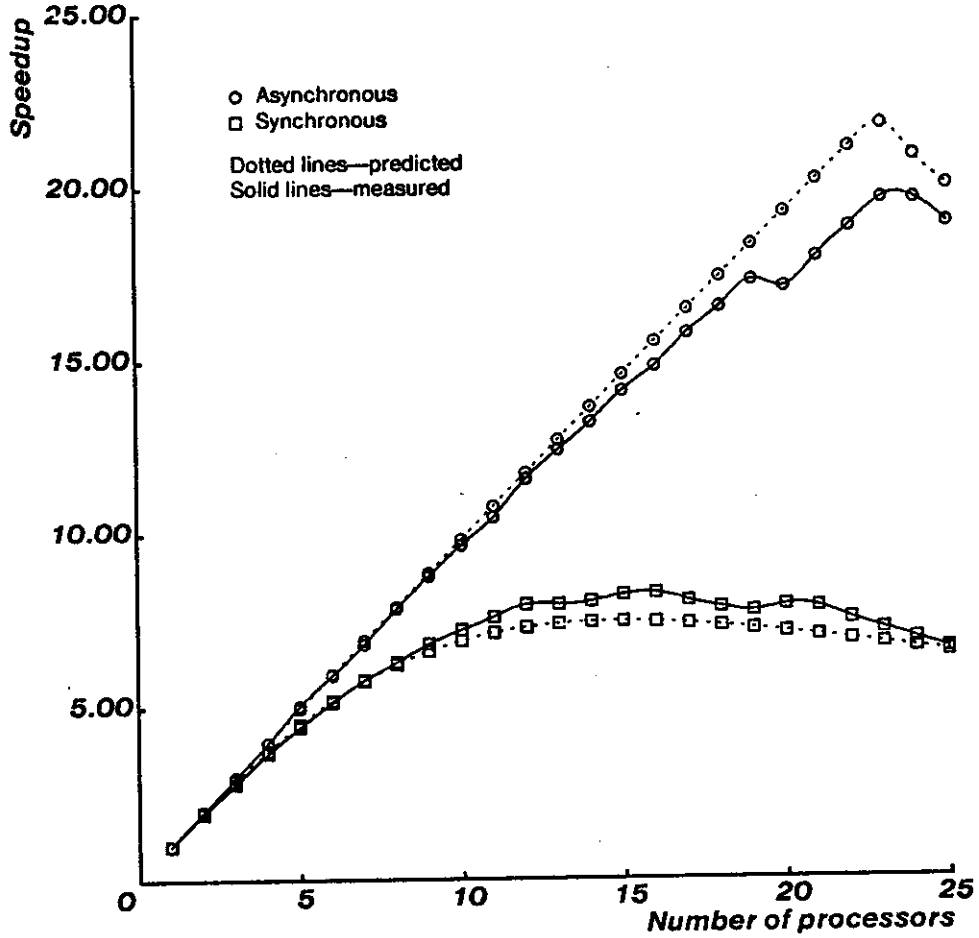

Figure 12: Influence of Synchronization on Molecular Dynamics Calculations

8.3 Matrix multiplication

From Section 5.1, the decomposition functions for matrix multiplication with local copies are—

$$f_p = \frac{(MT_1 + T_3)N}{MT_1 + T_3\sqrt{N}}$$

$$f_a = \sqrt{N}$$

Let us introduce

$$k_1 = \frac{T_1}{T_2} \qquad k_2 = \frac{T_1}{T_3} \tag{46}$$

to express the relationship between access and processing speeds without reference to particular hardware technologies.

We ran the local-copies algorithm on Cm*. The constants k_1 and k_2 were measured at 1.694 and 3.678 respectively. Correlation of the experimental and predicted curves is quite good, but falls off somewhat as more processors are added due largely to the undecomposable constant overhead added to each process by the need to perform loop initialization and to read the clock (the $T_{p_{fixed}}$ of Section 7). This overhead grows more significant as processors are added and the work per process declines. To try and factor out the effect of this overhead, which we call T_4, we solved the system of equations

$$T_4 + 48^3 T_1 = 82.3\ msec.\text{(measured processing time for } M = 48 \text{ without local copies)}$$

$$T_4 + 24^3 T_1 = 10.3\ msec.\text{(measured processing time for } M = 24 \text{ without local copies)}$$

The results of $T_1 = 7.4\times10^{-4}$ and $T_4 = 1.14\times10^{-2}$ "predicted" the execution time for $M = 36$ within 1%. From this, we derived the revised decomposition functions

$$t_p = T_4 + \frac{M^3 T_1}{N} + \frac{M^2 T_3}{\sqrt{N}} \qquad t_a = \frac{M^2 T_2}{N}$$

We can proceed as before to derive equations to predict speedup. These equations are graphed against the observed values in Figure 13. The measured values are everywhere within 5% of the predicted values. The close correspondence deteriorates slightly for increasing values of N. One reason is likely the fact that the initialization of the inner loop grows more significant as it gets shorter; for $M = 24$ and $N = 16$, the inner loop is executed only six times before it terminates.

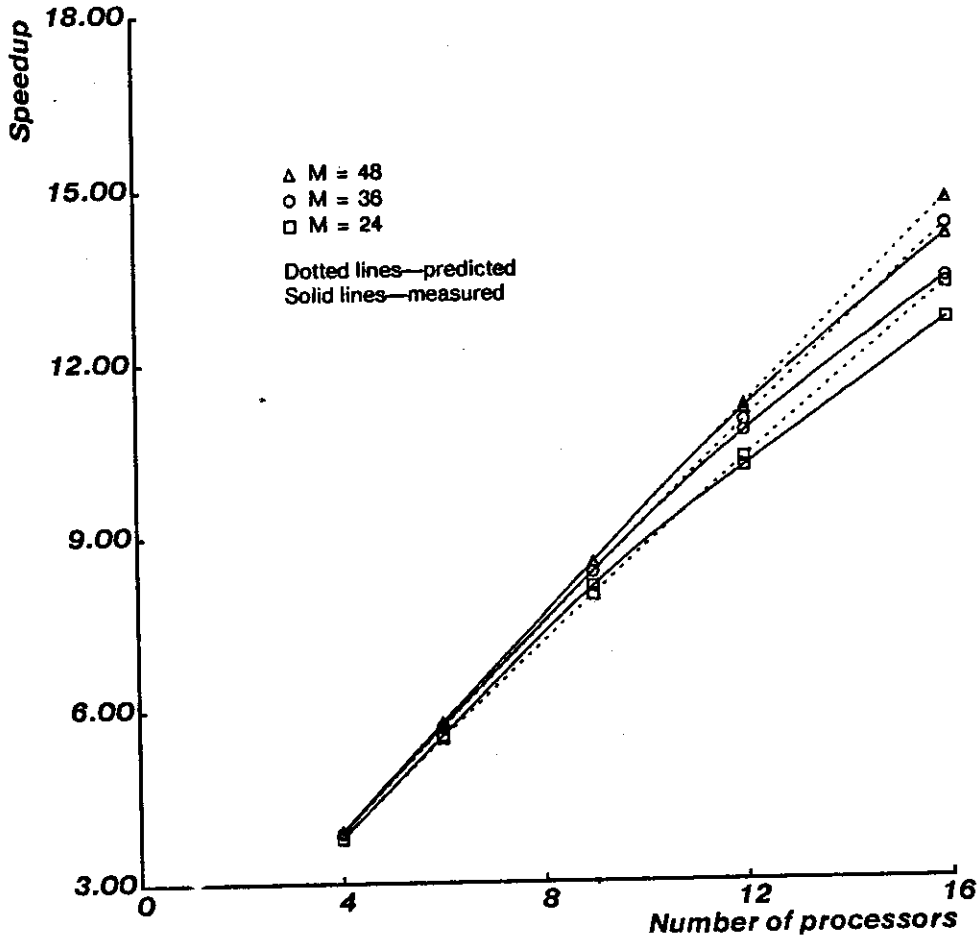

Figure 13: Speedup of Matrix Multiplication

9 Improving a Parallel Implementation: A Case Study

A major motivation of performance prediction is to narrow the space of experimental alternatives. As an example, we will present a case study based on the widely known algorithm for solving Poisson partial differential equations. This case study refines the analysis of this algorithm which has been presented in Section 5.2, which considered a decomposition of the grid into squares.

Data elements are global if they occupy positions on a boundary between partitions of the grid belonging to different processors. The access time t_a is proportional to the number of boundary elements. On the other hand, the processing time t_p is proportional to the surface—the number of elements in a partition. Hence, a decomposition which assigns a processor to a partition with a larger surface-to-boundary ratio will yield a larger X, and hence better performance.

For comparison, we will present two decompositions other than the square decomposition. Although all three decompositions are members of the $(N; \sqrt{N})$ group, their different processing-to-access ratios X and slight differences in decomposition functions produce slight differences in performance.

9.1 Decomposition into triangles

Let us assume that a surface is divided into triangular areas, each with base $a-1$ and height a. Then a square of size $a \times a$ can be divided into two triangles of the kind shown in Figure 14. (Actually, the regions are only "approximately" triangles, due to the inflection at the middle of the hypotenuse, an inflection which is necessary to allow two such regions to fit exactly into a square.) As in Section 5.2, elements on a boundary are stored in global memory. In order to partition the $M \times M$ grid into N triangles, it must be divided into $N/2$ two-triangle squares, each with dimension $a = \sqrt{2}\, M/\sqrt{N}$.

Without local copies:

As in Section 5.2, the number of global accesses required to update elements near the boundary depends on their position within the triangle. As before, each of these elements are classified according to the number of global accesses made to it per iteration. Figure 14 shows the class to which each element belongs. Let n_j denote the cardinality of the class j for an individual triangle, and t_{aj} the access time to update an element in class j.

Class 1: $t_{a1} = T_2$ $\quad n_1 = (a-6) + (a-5) = 2a-11$

Class 2: $t_{a2} = 2T_2$ $\quad n_2 = a-4$

Class 3: $t_{a3} = 3T_2$ $\quad n_3 = 2$

Class 4: $t_{a4} = 4T_2$ $\quad n_4 = a-5$

```
6 x x x x x x x x x x x x x x
6 5 x x x x x x x x x x x x x
5 3 4 x x x x x x x x x x x x
5 1 2 4 x x x x x x x x x x x
5 1 0 2 4 x x x x x x x x x x
5 1 0 0 2 4 x x x x x x x x x
5 1 0 0 0 2 4 x x x x x x x x
5 1 0 0 0 0 2 5 x x x x x x x
5 1 0 0 0 0 1 4 x x x x x x x
5 1 0 0 0 0 0 2 4 x x x x x x
5 1 0 0 0 0 0 0 2 4 x x x x x
5 1 0 0 0 0 0 0 0 2 4 x x x x
5 1 0 0 0 0 0 0 0 0 2 4 x x x
5 1 0 0 0 0 0 0 0 0 0 2 4 x x
5 2 1 1 1 1 1 1 1 1 1 3 5 x x
6 5 5 5 5 5 5 5 5 5 5 5 5 6 6 x
```

Figure 14: Triangular decomposition of a Poisson PDE

Class 5: $t_{a5} = 5T_2 \qquad n_5 = (a-4) + (a-3) + 3 = 2a - 4$

Class 6: $t_{a6} = 6T_2 \qquad n_6 = 5$

Based on the number of elements in each class, decomposition into triangles without local copies can be described by these equations. The constants k_1 and k_2 are as defined in (46).

$$t_p = \frac{M^2 T_1}{N}$$

$$t_a = \sum_{i=1}^{6} t_{ai} n_i = \left[\frac{18\sqrt{2}\,M}{\sqrt{N}} - 23\right] T_2 = [18\sqrt{2}\,M - 23]\,\frac{18\sqrt{2}\,M - 23\sqrt{N}}{(18\sqrt{2}\,M - 23)\sqrt{N}}\,T_2$$

$$T_p = M^2 T_1 \qquad T_a = (18\sqrt{2}\,M - 23)T_2$$

$$X = \frac{M^2 T_1}{(18\sqrt{2}\,M - 23)T_2} = \frac{M^2}{(18\sqrt{2}\,M - 23)}\,k_1$$

$$f_p = N$$

$$f_a = \frac{(18\sqrt{2}\,M - 23)\sqrt{N}}{18\sqrt{2}\,M - 23\sqrt{N}}$$

With local copies:

Each triangle contains $3a-4$ boundary elements. It takes $6a-8$ accesses to copy these elements back and forth. In addition, another $3a$ accesses are needed to read boundary elements from adjacent triangles. Thus, each process makes a total of $9a-8 = 9\sqrt{2}\,M/\sqrt{N}-8$ global accesses.

$$t_p = \frac{M^2T_1}{N} + \frac{9\sqrt{2}\,MT_3}{\sqrt{N}} - 8T_3 = T_p\frac{M^2T_1 + 9\sqrt{2}\,MT_3\sqrt{N} - 8T_3N}{(M^2T_1 + 9\sqrt{2}\,MT_3 - 8T_3)N}$$

$$t_a = \frac{9\sqrt{2}\,MT_2}{\sqrt{N}} - 8T_2 = [9\sqrt{2}\,MT_2 - 8T_2]\frac{9\sqrt{2}\,MT_2 - 8T_2\sqrt{N}}{(9\sqrt{2}\,MT_2 - 8T_2)\sqrt{N}}$$

$$T_p = M^2T_1 + 9\sqrt{2}\,MT_3 - 8T_3 \qquad T_a = 9\sqrt{2}\,MT_2 - 8T_2 \qquad X = \frac{M^2}{9\sqrt{2}\,M-8}k_1 + \frac{k_1}{k_2}$$

$$f_p = \frac{N(M^2T_1 + 9\sqrt{2}\,MT_3 - 8T_3)}{M^2T_1 + 9\sqrt{2}\,MT_3\sqrt{N} - 8NT_3}$$

$$f_a = \frac{(9\sqrt{2}\,M-8)\sqrt{N}}{9\sqrt{2}\,M-8\sqrt{N}}$$

Speedups for various dimensions of the element grid are shown in Figure 15. When k_2 is high, copying is relatively inexpensive, so it pays off, almost regardless of the number of processors N. With a low k_2, copying is not productive unless the number of processors is fairly high. Note that the non-copying version attains its maximum speedup at $N=31$, after which contention causes it to decline.

9.2 Decomposition into hexagons

Imagine that the grid is again divided into squares of dimension $a \times a$, with a being $2M/\sqrt{N}$. Let us now divide the grid into hexagons, each of which occupies one-fourth of a square, as shown in Figure 16. If we let

$$b = \frac{a}{4} = \frac{M}{2\sqrt{N}}$$

then the access and processing times can be calculated from the number of atomic accesses for each element given in Figure 16.

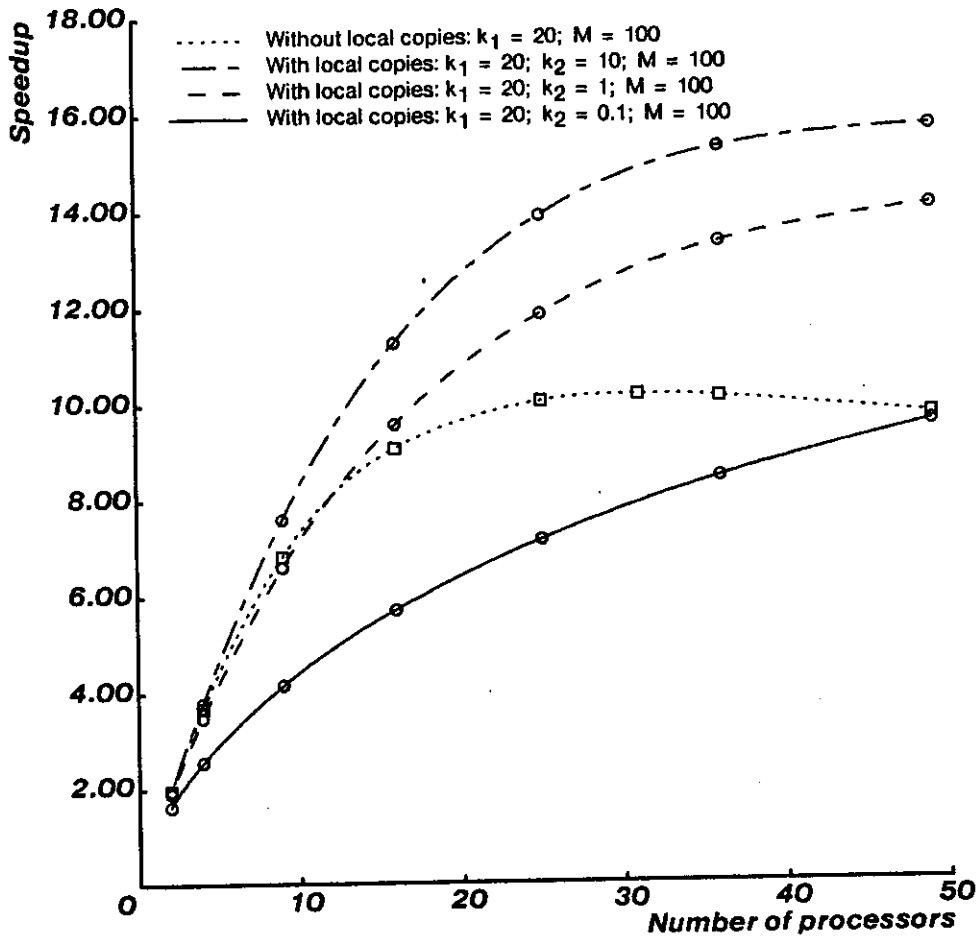

Figure 15: Speedup versus N for triangular decomposition of a Poisson equation

Without local copies:

Class 1: $t_{a1} = T_2$ $n_1 = 2(b-1)$

Class 2: $t_{a2} = 2T_2$ $n_2 = 2(b-2) + 2(b-1)$

Class 3: $t_{a3} = 3T_2$ $n_3 = 2$

Class 4: $t_{a4} = 4T_2$ $n_4 = 2(b-2) + 2(b-1)$

Class 5: $t_{a5} = 5T_2$ $n_5 = 2(b+2)$

Based on the number of elements in each class, the decomposition functions can be calculated as:

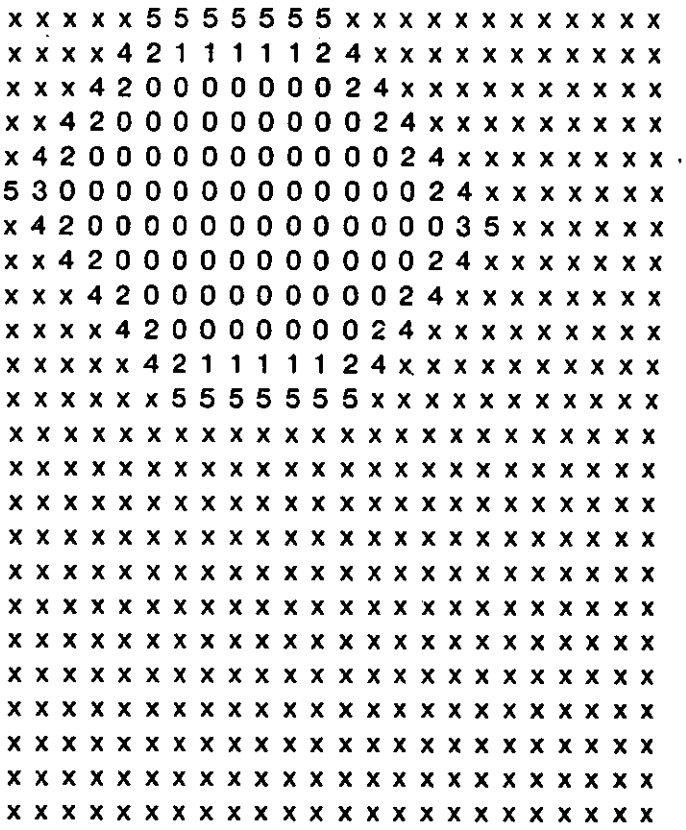

Figure 16: Hexagonal decomposition of a Poisson PDE

$$t_p = \frac{M^2 T_1}{N} \qquad t_a = \sum_{i=1}^{5} t_{ai} n_i = \frac{18MT_2}{\sqrt{N}} - 12T_2 = 6(3M-2)T_2 \frac{3M-2\sqrt{N}}{(3M-2)\sqrt{N}}$$

$$T_p = M^2 T_1 \qquad T_a = 6(3M-2)T_2 \qquad X = \frac{M^2}{6(3M-2)} k_1$$

$$f_p = N \qquad f_a = \frac{(3M-2)\sqrt{N}}{3M-2\sqrt{N}}$$

With local copies:

The hexagon has six sides. If the two horizontal sides are considered to have $b + 1$ elements, then the other four sides each consist of $b - 1$ elements, for a total of $6b - 2$ boundary elements, each of which must be copied back and forth. Also, $2(b + 1) + 4b$ elements from neighboring hexagons will be referenced, for a total of $18b - 2$ nonlocal references.

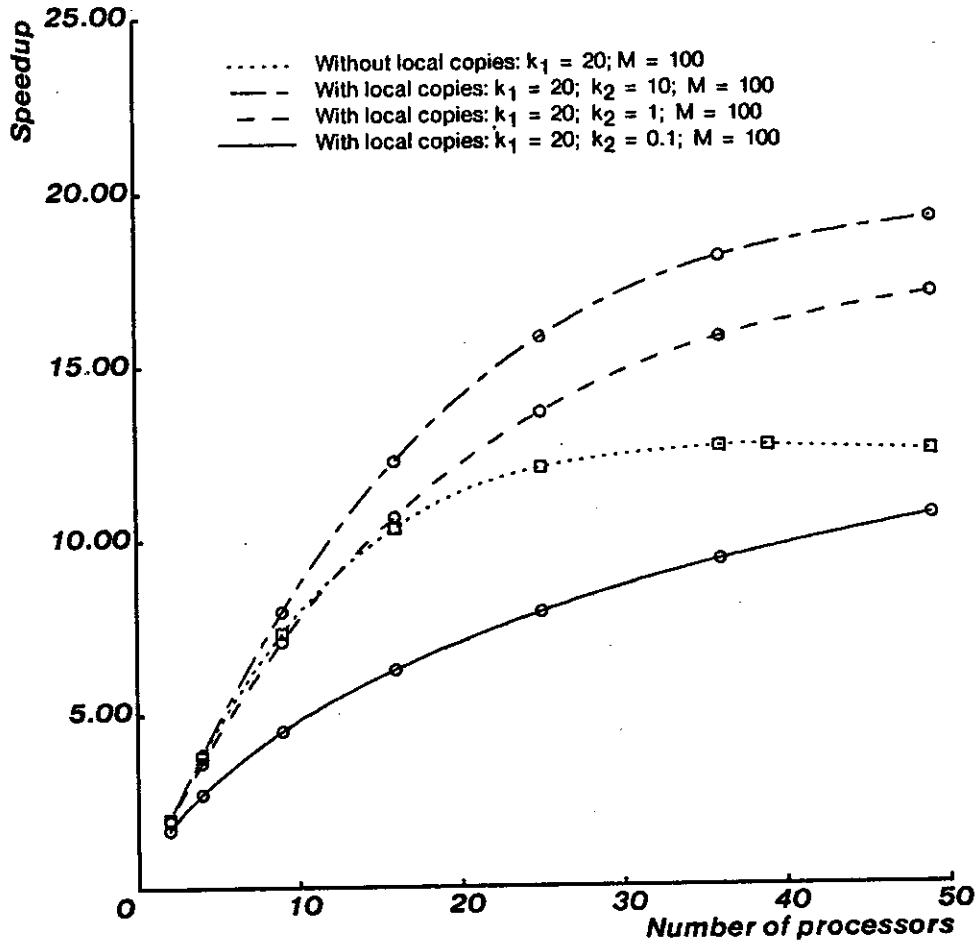

Figure 17: Speedup versus N for a hexagonal decomposition of a Poisson equation

$$t_p = \frac{M^2T_1}{N} + \frac{9MT_3}{\sqrt{N}} - 2T_3 = T_p \frac{M^2T_1 + 9MT_3\sqrt{N} - 2T_3N}{(M^2T_1 + 9MT_3 - 2T_3)N}$$

$$t_a = \frac{9MT_2}{\sqrt{N}} - 2T_2$$

$$T_p = M^2T_1 + 9MT_3 - 2T_3 \qquad T_a = 9MT_2 - 2T_2 \qquad X = \frac{M^2k_1}{9M-2} + \frac{k_1}{k_2}$$

$$f_p = \frac{(M^2T_1 + 9MT_3 - 2T_3)N}{M^2T_1 + 9MT_3\sqrt{N} - 2T_3N} = \frac{(M^2k_2 + 9M - 2)N}{M^2k_2 + 9M\sqrt{N} - 2N}$$

$$f_a = \frac{(9M-2)\sqrt{N}}{9M - 2\sqrt{N}}$$

The speedup versus the number of processors for various grid dimensions M is shown in Figure 17. Again, a lower k_2 induces a greater speedup, and the non-copying implementation reaches a

maximum speedup, this time at $N = 39$. Notice that whether or not copies are made, for all values of N, the hexagonal decomposition exhibits a better speedup than the triangular decomposition, owing to its lower boundary-to-surface ratio.

A square has a boundary-to-surface ratio smaller than a triangle but larger than a hexagon. (For example, a 144-element hexagon would have 34 boundary elements, a 144-element square would have 44, and a hypothetical 144-element right triangle would have approximately 46.9.) Thus we would expect a square decomposition to show a speedup somewhere between those shown by triangular and hexagonal decompositions. Figures 18 and 19 show that this is indeed the case.

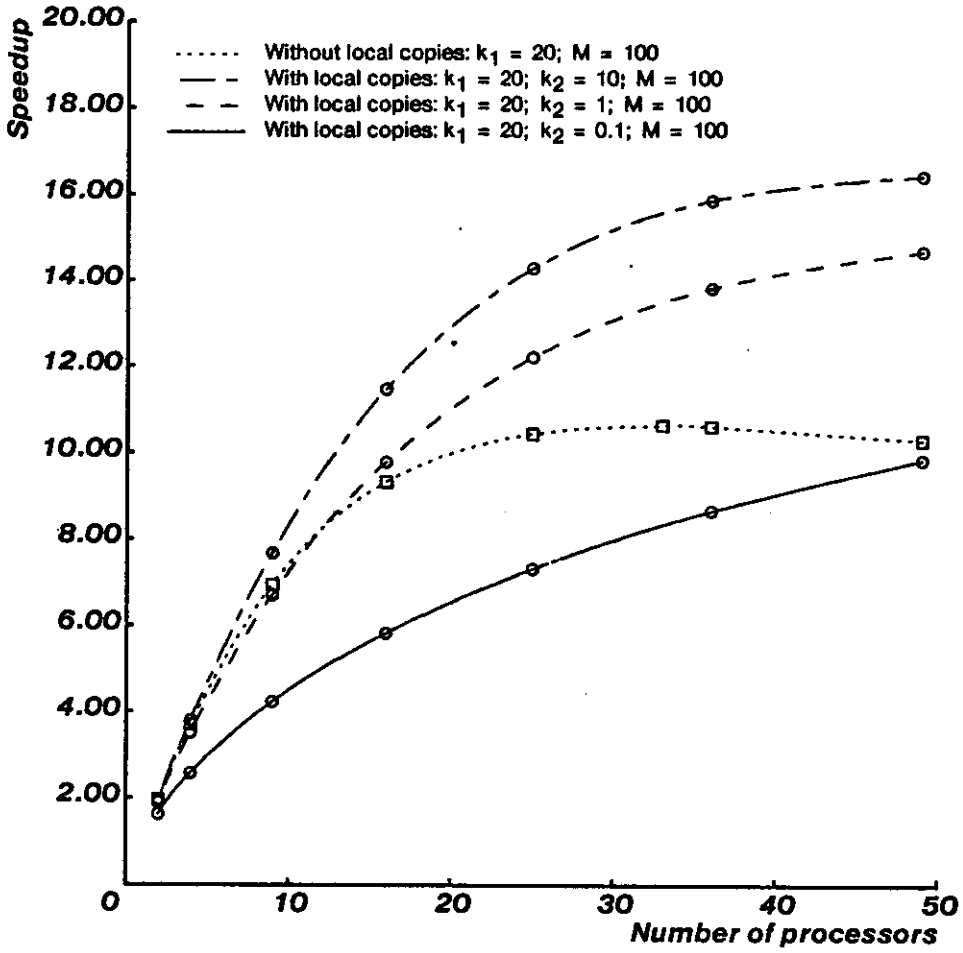

Figure 18: Speedup versus N for a square decomposition of a Poisson equation

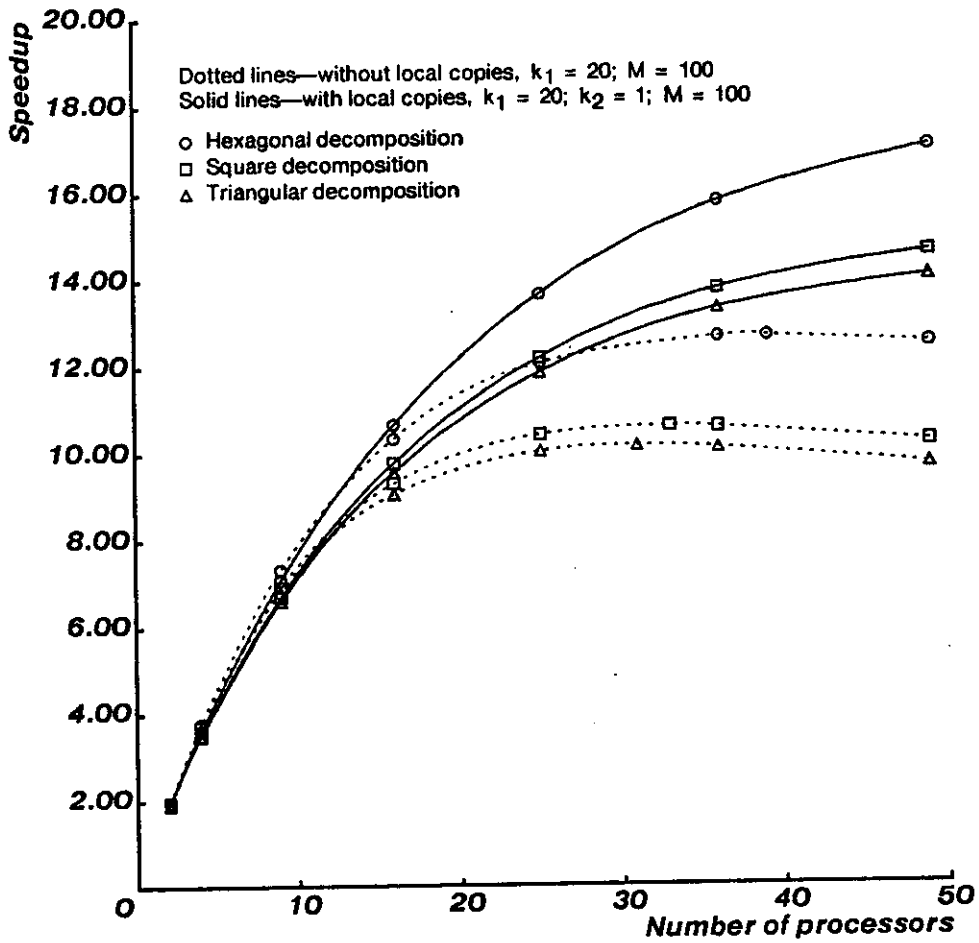

Figure 19: Speedup versus *N* for triangular, square, and hexagonal decompositions

10 Conclusions

We have defined a simple multiprocessor model, which was then enhanced to accommodate processors and memories of different speeds. The model has shown that improving processor or memory speed is only effective up to a point. The model bounds the worst performance that can be expected from a synchronous algorithm and predicts the best performance that can be achieved by an asynchronous one.

The performance of the multiprocessor implementation of an algorithm is frequently expressed in terms of speedup. Speedup is a function of the way an algorithm is decomposed. We have defined decomposition functions for the processing and access times, which tell how the per-process times change as the number of processors is increased. Algorithms can be divided into decomposition groups based on these functions—groups which display characteristic speedup curves for varying numbers of processors.

Several multiprocessor algorithms have been studied in detail; some of them have been implemented on the Cm* multiprocessor. Measurements have been made for differing decomposition strategies and numbers of processors.

The experimental measurements show that even a simple analytical model can be a solid base for a multiprocessor performance prediction. Since the model parameters represent real workload characteristics, rather than some average values expressed by probabilities, they give us the opportunity to evaluate the sensitivity of performance to different parameters independently. Due to the factors enumerated in Section 7, the measured performance sometimes differs significantly from the predicted one, but the correlation between the modeled and the measured curves is obvious.

Although the simple analytical model presented here gives satisfactory results, there are several areas for improvement.

- It was shown that the upper and lower bounds could be predicted if a workload with only one atomic (t_p, t_a) pair is assumed. Can operators and an associated calculus be found, such that when applied to a workload, characteristic times will give a resulting (t_p, t_a) pair which predicts the performance of a parallel workload more closely than the original model?

- Sometimes better performance can be achieved if each process makes local copies of global data. Even better performance could be achieved if the amount of local processing (T_3) needed to produce these copies is kept low. Hence, the possibility of providing architectural support for global-to-local (and local-to-global) data transitions should be explored. For example, broadcasting global data could facilitate the generation of local copies.

- In order to make the t_w to N dependency linear, circuit-switching interconnection networks are assumed in this paper. Models should be provided for the cases where this dependency is nonlinear.

- It was also assumed that all the work of executing a parallel program can be neatly decomposed into N parallel processes. It is more likely that some fixed amount of local processing and/or global access will have to be done in every process anyway. Therefore, decomposition functions other than

$$t_x = \frac{T_x}{F_x}$$

should be investigated.

- Several classic parallel workloads have been measured. However, comparisons are often difficult due to implementation differences. Classification of these workloads into decomposition groups would make it easier to compare them. In this way, benchmarking could be done on the basis of typical decompositions rather than typical algorithms.

Multiprocessors bring a new dimension to the already very complex experimentation space of uniprocessors. Only good models accompanied by structured programming can narrow this space. The multiprocessor-system design cycle can be shortened only if experiments are carefully selected with the assistance of a performance model, and measured results are analytically identified.

Acknowledgments

The authors would like to thank David Black for an improved version of the proof in Section 2.5.

References

1. Marsan, M. A. and Gerla, M., "Markov Models for Multiple Bus Multiprocessor Systems," *IEEE Trans. on Computers,* Vol. C-31, No. 3, March 1982, pp. 239 – 248.

2. Baskett, F. and Smith, A.J., "Interference in Multiprocessor Computer Systems and Interleaved Memory," *Comm. ACM,* Vol. 19, No. 6, June 1976, pp. 327 – 334.

3. Bhandarkar, D.P., "Analysis of Memory Interference in Multiprocessors," *IEEE Trans. on Computers,* Vol. C-24, No. 9, September 1975, pp. 897 – 908.

4. Marsan, M.A. and Gregoretti, F., "Memory Interference Models for a Multimicroprocessor Model with Shared Bus and Single External Common Memory," *Euromicro J.,* February 1981, pp. 124 – 133.

5. Vrsalovic, D. and Siewiorek, D.P., "Performance Analysis of Multiprocessor Based Control Systems," *Proceedings of the Real-Time Systems Symposium,* December 1983, pp. 73 – 78.

6. Kruskal, C.P., "Searching, Merging, and Sorting in Parallel Computation," *IEEE Trans. on Computers,* Vol. C-32, No. 10, October 1983, pp. 942 – 946.

7. Whiteside, R., Hibbard, P., and Ostlund, N., "A Case Study in the Application of a Tightly Coupled Multiprocessor to Scientific Computations," in *Parallel Computations,* Rodrigue, G., ed., Academic Press, 1982, pp. 315 – 364.

8. Hoshino, T. and Shirakawa, T., "Load Flow Simulation of Three-Dimensional Boiling Water Reactor Core by PACS-32 Parallel Microprocessor System," *Nuclear Technology,* Vol. 56, No. 3, March 1982, pp. 465 – 477.

9. Hoshino, T., Kawai, T., Shirakawa, T., Higashino, J., Yamaoka, A., Ito, H., Sato, T. and Sawada, K., "PACS: A Parallel Microprocessor Array for Scientific Calculations," *ACM Transactions on Computer Systems,* Vol. 1, No. 3, August 1983, pp. 203 – 221.

10. Baer *et al.*, "Binary Search in a Multiprocessing Environment," *IEEE Trans. on Computers,* Vol. C-32, No. 7, July 1983, pp. 667 – 677.

11. Jones, A.K. and Gehringer, E.F. editors, "The Cm* Multiprocessor Project: A Research Rewiew," Tech. report, Carnegie-Mellon University, 1980.

12. Jones, A. and Schwarz, P., "Experience using multiprocessor systems—a status report," *ACM Computing Surveys,* Vol. 12, No. 2, June 1980, pp. 121 – 165.

13. Whiteside, R., Hibbard, P., and Ostlund, N., "Conventional and Systolic Parallel Algorithms for Monte Carlo Simulations of Molecular Motion", Submitted to *ACM Transactions on Computer Systems.*

14. Allen, A. O., *Probability, Statistics, and Queueing Theory with Computer Science Applications,* Academic Press, 1978.

15. Singh, A. and Segall, Z., "Synthethic Workload Generation for Experimentation with Multiprocessors," Tech. report, Carnegie-Mellon University, 1982.

16. Vrsalovic D., Siewiorek D.P., Segall, Z. and Gehringer, E., "The Influence of Parallel Decomposition Strategies on the Performance of Multiprocessor Systems", submitted to the *International Conference on Fifth Generation Computer Systems,* Tokyo, Nov. 6 – 9, 1984.

17. Hockney, R.W., "Characterizing Computers and Optimizing the FACR(l) Poisson-Solver on Parallel Unicomputers," *IEEE Trans. on Computers,* Vol. C-32, No. 10, October 1983, pp. 933 – 941.

18. Siomalas, K.O. and Bowen, B.A., "Performance of Cross-Bar Multiprocessor Systems," *IEEE Trans. on Computers,* Vol. C-32, No. 7, July 1983, pp. 689 – 695.

19. Lang, T., Valero, M. and Fiol, M.A., "Reduction of Connections for Multibus Organization," *IEEE Trans. on Computers,* Vol. C-32, No. 8, August 1983, pp. 707 – 715.

20. Vaughan, R.F. and Anastas, M.S., "Limiting Multiprocessor Performance Analysis," *Proceedings of the Conference on Parallel Processing,* 1979.

21. Korn, D., "Timing Analysis for Scientific Codes Run Under Washcloth Simulation," Ultracomputer Note # 24, Courant Institute, N.Y.U., 1981.

22. Lavenberg, S. editor, *Computer Performance Modeling Handbook,* Academic Press, Notes and Reports in Computer Science and Applied Mathematics, 1983.

23. Ferrari, D., *Computer Systems Performance Evaluation,* Prentice-Hall, 1978.

24. Vrsalovic, D., Siewiorek, D., Segall, Z., and Gehringer, E., "Performance prediction for multiprocessor systems," 13th International Conference on Parallel Processing, Bellaire, MI, Aug. 21 – 24, 1984 (to appear)