1992

Speedup, Communication Complexity and Blocking - A La Recherche du Temps Perdu

Dan C. Marinescu

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
92-057

SPEEDUP, COMMUNICATION COMPLEXITY AND BLOCKING-A LA RECHERCHE DU TEMPS PERDU

Dan C. Marinescu
John R. Rice

CSD-TR-92-057
August 1992

Speedup, Communication Complexity and Blocking — A La Recherche du Temps Perdu*

Dan C. Marinescu and John R. Rice
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

September 8, 1992

Abstract

The paper investigates the time lost in a parallel computation due to sequential and duplicated work, communication and control, and blocking. It introduces the concept of relative speedup and proposes characterizations of parallel algorithms based upon the communication complexity and the blocking model. The paper discusses the impact of the processor's architecture upon the measured speedup. It shows that a large speedup may be due to an inefficient sequential computation, e.g., due to the cache management, rather than to an efficient parallel computation.

A model of parallel computations which takes into account sequential and duplicated work, communication and control and blocking is presented. The paper shows that the scalability of a parallel computation is determined by the communication complexity. The model is used to predict the asymptotic behavior, the maximum speedup and the optimal number of processors. An incore 3-D FFT algorithm for distributed memory MIMD systems and a Chebychev iterative algorithm for solving a linear system of equations are used to illustrate the concepts introduced in this paper.

*Work supported in part by a grant from the National Science Foundation, CCR-9119388

1 Introduction

This paper investigates the time lost in a parallel computation due to different sources of inefficiency. Sequential and duplicated work, communication and control, and blocking are the major factors limiting the performance of a parallel computation. This paper is primarily concerned with control parallel algorithms and programs for MIMD systems. The computational model discussed considers P *threads of control* running concurrently and communicating with one another through abstractions called *communication channels.* Any interruption of the flow of control of any thread for communication and/or control is called an *event.*

The model is mapped to a particular MIMD architecture by associating threads of control with processing elements, PEs, and communication channels with communication hardware/software. Events correspond to sending/receiving of messages for the message passing model suitable for distributed memory MIMD systems, or to access to shared data in a shared memory model for a shared memory MIMD system.

Once the computational model is mapped to a MIMD architectures, one can make statements concerning the computing time, the work required to carry out the computation, the work intensity or the instruction execution rate, and about the performance of the ensemble consisting of the parallel algorithms, its implementation and the parallel architecture.

This paper discusses several experimental and theoretical questions pertinent to the performance characterization and analysis of parallel computations on MIMD systems. First it addresses the problem from the point of view of a practitioner and shows that measuring the speedup is often a difficult, if not impossible task, due to time and space limitations of the sequential computation. Furthermore, the results of a speedup measurement can be misleading. Instead of reflecting an efficient parallel computation, a large speedup may be due to an inefficient sequential computation.

The concept of relative speedup introduced in §2 reflects the fact that space and time requirements impose a minimum number of PEs, $P_{min} > 1$ for massively parallel computations, and that it only makes sense to compare the execution times of a parallel computation over a range of PEs, ($P_{min} \leq P \leq P_{max}$). Communication complexity, the blocking model and the lifetime of intermediate results are suggested as means to provide an architecture independent characterization of a parallel algorithm [8]. A model of parallel execution which takes into account the effects of sequential and duplicated work, communication complexity and blocking is presented in §3. An incore 3-D FFT algorithm for a hypercube is discussed in §4 and an experiment in monitoring a Chebychev iterative algorithm for solving a linear system of equations is presented in §5.

2 The Speedup and Other Measures of Performance

The *speedup* of a parallel computation on a multiprocessor with P identical processing elements is the ratio

$$S(P) = \frac{T(1)}{T(P)}$$

with $T(1)$ the execution time of a serial implementation and $T(P)$ the one of a parallel implementation using P processing elements. The *speedup curve* $S = S(P)$ is the graph of the speedup as a function of P, the number of PEs.

The speedup is considered a simple and expressive way to measure the overall performance of a parallel algorithm and of its implementation on a parallel architecture. The speedup appeals to the practitioner interested in a simple way to measure the performance of a parallel application and to those interested in more theoretical aspects of parallel algorithm design, parallel architectures, and performance. The speedup provides an elegant way to reason about the asymptotic behavior of a parallel computation, and upper bounds for the speedup can be derived in the framework of a theoretical model [1], [8]. In the same time, it seems relatively easy to measure accurately the speedup of any application.

Yet the virtues of the speedup concept must be reexamined as our understanding of parallel computing and our experience in using massively parallel computers grow. The question whether the problem size should be allowed to grow when the number of PEs increases or if fixed-size problems should be considered, has lead to the introduction of the concept of the *scaled speedup* [2]. Allowing the size for a problem to grow subject to an upper bound on the execution time, produces different results than considering space (memory) constraints as pointed out in [11].

The scaled speedup addresses the concern that the parallel execution time, the efficiency, and the speedup for a small, fixed size problem, can only be very small when the number of PEs allocated to the problem exceeds a certain range. But the converse is also true. Given large, fixed size problems suitable for a configuration with a large number of PEs, the execution time of the corresponding sequential computation can only be very large. Therefore, for massively parallel computations, it seems reasonable to define a range of P_{min} to P_{max} processing elements and compare the execution time over that range only.

From a practical standpoint, it is very difficult and often impossible to measure the speedup of very large problems running on existing MIMD systems. MIMD systems with hundreds of PEs are in use today and promising results for different applications running on such systems are reported. For example, applications requiring $10^4 - 10^5$ seconds on a 520 node Touchstone Delta are described [12]. The performance of such computation is given in

terms of the Mflops rate rather than in terms of the speedup, simply because the execution time of a sequential implementation is prohibitively large and cannot be measured.

Even for small problems, it is often impossible to measure the execution time of the sequential implementation due to storage constraints. One of the main attractions of distributed memory MIMD systems is the large amount of storage available. It is very unlikely that an application which needs all the 8 Gbytes of storage available on 520 PEs of the Touchstone Delta (or 32 Gbytes on a 1000 PE CM5), is able to run in the 16 Mbytes available on a single node of the Delta (or 32 Mbytes on a node of CM5). In Section 4, we discuss the performance of a 3-D FFT computation on an INTEL iPSC/860 with 16 Mbytes/node. The measurements show that, even for relatively modest sizes of a 3-D mesh, the computation cannot be carried out in one node only, due to storage limitations. The dimension of the mesh considered were in the 16 to 8192 range. In about 30% of the cases (70 out of 210 different mesh sizes), the speedup could not be determined because the problem was too large to run in a single node.

A further complication is due to the fact that the instruction execution rate (the Mflops rate) of modern RISC processors like the i860 used to build massively parallel systems, is very sensitive to the cache management. Experiments reported in [3] show that, depending upon the cache usage, the BLAS routines run at rates ranging from 10 Mflops to the peak rate of 60 Mflops. Experiments with the 3-D FFT computation, discussed in Section 4, show that the Mflops rates for a problem of constant size, but with different mesh shapes differ by a factor of as much as 2.5. Yet the speedup is based upon the implicit assumption that the sequential and the parallel implementation used to measure the speedup run on a computing engine with a fixed instruction execution rate. If the instruction rate is not constant, the speedup may appear artificially high, even though the aggregate Mflops rate is low, simply because the sequential implementation is inefficient. In Section 4, it is shown that a 3-D FFT computation of constant size (262144 nodes), runs at only 75 Mflops and exhibits a speedup of about 10 on a 16 node iPSC/860 when the mesh shape is $16 \times 16 \times 1024$, but runs at about 125 Mflops with a speedup of only 7 for a $64 \times 64 \times 64$ mesh. To account for this effect, the *speedup inflation rate*, $infl(P) = \frac{I(P)}{I(1)}$, is introduced and it is suggested that the *effective speedup* $S^{ef}(P) = \frac{S(P)}{infl(P)}$ be used instead of the speedup. In this expression $I(P)$ represents the instruction execution rate with P processing elements and $I(1)$ the rate when only one PE is used.

To provide a practical measure of performance for problems of growing size, the concept of *relative speedup* is introduced. Let Q be the smallest number of PEs that can be used to

run a problem of fixed size. The relative speedup $S_{P,Q}$ is defined as

$$S_{P,Q} = \frac{T(Q)}{T(Q+P)}.$$

Clearly $S_{P,Q} = \frac{S(Q+P)}{S(Q)}$. The relative speedup curve is then the function

$$S_{P,Q} = S_{P,Q}(P).$$

While the relative speedup provides a practical measure of performance useful for a practitioner interested in performance analysis, we propose to investigate other measures of performance for the parallel algorithms, the implementation and their suitability to a particular parallel architecture.

The communication complexity relates the total number of events, E, to the number of threads of control, P; it provides an architecture independent measure of performance of a parallel algorithm. There are *embarrassingly parallel algorithms* with $E = \mathcal{O}(P)$ used in image processing and other applications like the electron density averaging for computations of macromolecular structures [10]. The effects of the communication complexity upon the asymptotic behavior of the speedup are discussed in §3.2.

Another characterization of a parallel algorithm reveals the type of blocking experienced. An *asynchronous algorithm* has the potential of keeping all the threads of control running at all times, therefore the blocking time, T_{blk} (the time one PE spends waiting on results from another PE) is zero. Algorithms which require *global synchronization* can be modeled as having $T_{blk} = \mathcal{O}(P)$. For example, iterative methods typically require algorithms in this class [9]. Even though the communication complexity required by global synchronization could be reduced to $E = \mathcal{O}(P)$ such computations can be rather inefficient if the blocking times are large. When global synchronization is achieved through broadcasting, as in the example discussed in §5, then $E = \mathcal{O}(P^2)$.

Yet another characterization of a parallel algorithm important for implementation on distributed memory MIMD systems, is the *lifetime of intermediate results.* The message passing programming model is where a thread of control produces intermediate data, performs an explicit action to send these data to other thread(s) which consume the data. To avoid blocking phenomena, namely consumer threads waiting for the data to be produced, an algorithm may attempt to produce the intermediate results as soon as possible. But in this case, the intermediate data must be buffered either at the sender's site, at the consumer's site, or within the system communication network. In case of massively parallel computations, intermediate results are produced at a high rate, due to the large number of threads

of control, and, if their lifetime is significant, then all the storage space is exhausted and the computation deadlocks.

The 3-D FFT computation discussed in §4, avoids this type of problem and ensures that data is consumed as soon as it is produced. The strategy is to group the threads of control into pairs, to have both PEs allocated to threads in the same pair first act as consumers by issuing an asynchronous *receive*, then synchronize and finally send the data. In this case, the number of outstanding messages is zero. At the other end of the spectrum are algorithms with P threads of control broadcasting data, potentially at the same time, so that each PE may have $(P-1)$ outstanding messages at some time and the total number of outstanding messages may be $P(P-1)$. We propose to measure the lifetime of intermediate results by the maximum number of outstanding messages at any one time.

3 A Model of Parallel Execution and Upper Bounds of the Speedup

A model of parallel computation is discussed in this section which is able to predict the asymptotic behavior, the maximum value for the speedup, and the optimum number of PEs. The model takes into account different sources of inefficiency in parallel computing. The effects of strictly sequential and duplicated work, the effect of the communication complexity and of blocking are discussed.

3.1 Sequential and Duplicated Work

Amdahl has shown that only rarely the entire work required by a computation can be carried out in parallel. If s denotes the fraction of the strictly sequential computation, Amdahl's law [1] shows that an upper bound for the asymptotic speedup is $S \leq 1/(1+s)$ and that only a relatively few PEs are needed to achieve the maximum speedup.

In addition to the strictly sequential work, there is an additional obstacle in reaching a large speedup when parallelizing a computation, namely some work needs to be duplicated by several or all threads of control in order to reduce the amount of communication and/or blocking. Computation of the trigonometric functions for a Fourier expansion is a good example of work duplicated in parallel FFT algorithms.

In the model presented in this paper, f denotes the fraction of the strictly sequential work, plus the fraction of the work duplicated by all the threads, and $W_{sd}(P)$ the total amount of additional work due to duplication and strictly sequential computation. Then $W_{sd}(P)$ reflects the fact that each of the $(P-1)$ threads wastes $f \times W(1)$ cycles, either

by being idle when only one thread carries out the strictly sequential computation or by replicating the work of one thread. $W(1)$ denotes the work carried out when only one thread of control is running. Then we have

$$W_{sd}(P) = (P-1) \times f \times W(1).$$

It follows that when only duplicated and sequential work is taken into account the total work with P threads is

$$W(P) = W(1) + W_{sd}(P) = W(1) \times (1 + f \times (P-1)).$$

We assume that the relationship between the execution time, $T(P)$, the total amount of work or computing cycles, $W(P)$, and the work intensity or instruction execution rate I is

$$W(P) = P \times I \times T(P) \quad \text{when} \quad P \geq 1$$

and it follows immediately that

$$S(P) = \frac{T(1)}{T(P)} = P \frac{1}{1 + (P-1)f}.$$

This is precisely the Amdhal's law, but here f represents the fraction of both sequential *and* duplicated work. To reach a fraction q of the asymptotic speedup, $S_\infty = 1/f$, $(0 \leq q \leq 1)$ then P_q processing elements are necessary with

$$P_q = \frac{q}{1-q} \frac{1-f}{f}.$$

For example when $f = 0.01$, the asymptotic speedup is $S_\infty = 100$. A speedup of 80 ($q = 0.8$) can be reached with $P_{0.8} = 400$ processing elements.

3.2 Communication complexity and the asymptotic speedup

Consider a parallel computation with P threads of control which need to communicate with one another. Call any interruption of a thread of control for communication and control an *event*, and denote the total number of events by E, and the average amount of work associated with an event by $\bar{\theta}$. The additional work for communication and control, $W_{cc}(P)$ is then given by $W_{cc}(P) = \bar{\theta}E$. If there is no sequential or duplicated work, and if no thread is idle during the entire computation, then

$$W(P) = W(1) + W_{cc}(P).$$

Following the same arguments concerning the time-work relationship, it follows that

$$S(P) = \frac{P}{1 + \alpha' E}$$

with

$$\alpha' = \frac{\bar{\theta}}{W(1)}.$$

When $E = kP$, then the asymptotic speedup is

$$S_\infty = \frac{1}{\alpha} \text{ with } \alpha = \frac{k\bar{\theta}}{W(1)}.$$

Two more cases are considered now, namely $E = kP \log P$ and $E = kP^2$. In both cases, the speedup reaches a maximum $S_{max} = S(P_{opt})$ and then goes to zero asymptotically. The maximum speedups are, respectively,

$$S_{max}^{(P \log P)} = \frac{1}{\alpha(1 - \log \alpha)} \text{ for } P_{opt} = \frac{1}{\alpha}$$

and

$$S_{max}^{(P^2)} = \frac{1}{2\sqrt{\alpha}} \text{ for } P_{opt} = \frac{1}{\sqrt{\alpha}}.$$

The speedup $S(P)$ with P processing elements and the *efficiency*, $\eta(P)$, are related by

$$S(P) = \eta(P) \times P.$$

It follows that in the two cases examined above, the efficiencies are, respectively,

$$\eta_{opt}^{(P \log P)} = \frac{1}{1 - \log \alpha} \text{ and } \eta_{opt}^{(P^2)} = \frac{1}{2}.$$

The effect of the communication complexity upon the speedup curve is shown in Figure 1.

3.3 The instruction execution rate and the speedup

Whenever the speedup of a particular computation is determined experimentally, it is implicitly assumed that the instruction execution rate is the same when $T(1)$ and $T(P)$ are measured. The model presented so far in this paper is based on the same reasonable assumption, namely that

$$W(1) = I \times T(1) \text{ and } W(P) = P \times I \times T(P) \text{ for } P > 1.$$

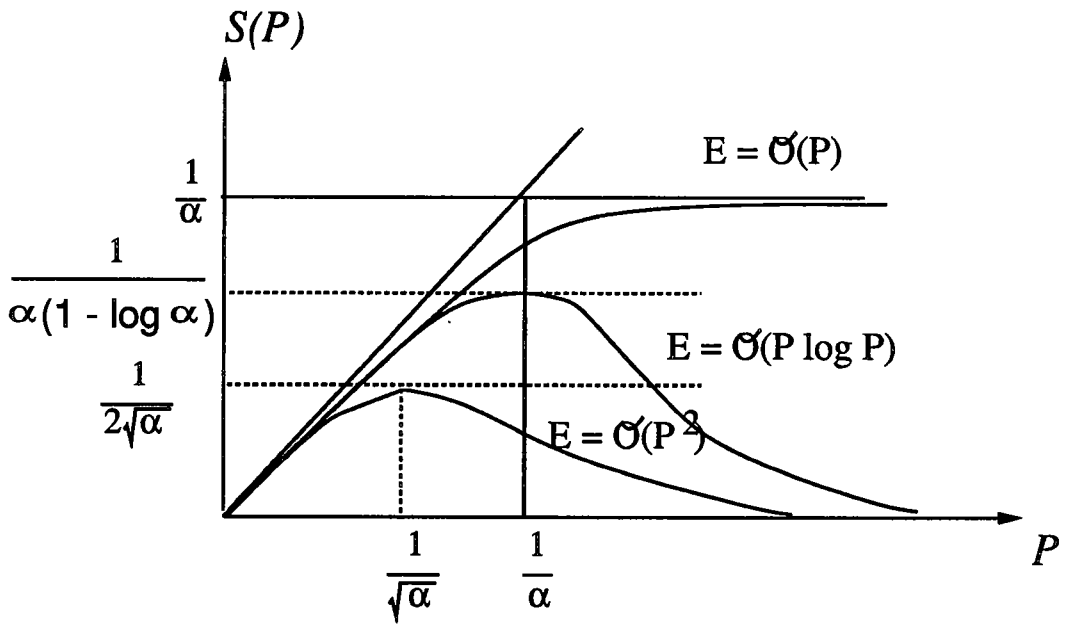

Figure 1: The speedups $S_{(P)}^{(P)}$, $S_P^{(P\log P)}$ and $S_{(P)}^{(P^2)}$ for the three cases $E = \mathcal{O}(P)$, $E = \mathcal{O}(P\log P)$ and $E = \mathcal{O}(P^2)$.

It makes little sense to talk about the speedup of an 8 processor CRAY Y-MP versus a 1 processor RS 6000 workstation. Yet this assumption may prove to be false and measurements reporting spectacular speedups may be misleading.

A first indication that this assumption may be false comes from an experiment involving a 3-D FFT on a mesh with $N = n_x \times n_y \times n_z$ grid points. This computation is described in some detail in Section 4.

To illustrate this effect a very simple experiment and its results are outlined below. The experiment consists of running a problem of fixed size on a variable number of PEs. The problem is to transform a vector of fixed length and use the **saxpy** BLAS routine available from the iPSC/860 math library. Table 1 presents the results.

Vector length	# of iterations	Execution time		Mflops rate per PE		Speedup
		1 PE (seconds)	8 PEs (seconds)	1 PE	8 PEs	
200,000	1,000	19.80	2.44	10.10	10.24	8.11
40,000	5,000	19.70	1.68	10.15	15.60	12.31
5,000	40,000	12.77	1.68	15.63	14.02	7.17

Table 1: Execution time, Mflops rate and speedup for a linear algebra computation.

Cases 1 and 2 which correspond to vector length of 200,000 and 40,000 respectively show a superlinear speedup. The higher speedup, 12.31 occurs when each of the 8 PEs processes vectors of length 5,000 and has a very high cache hit ratio. When the problem runs in 8 nodes a peak rate of 15.60 Mflops is achieved while the one PE case runs at about 10.15 Mflops.

The third case allows both the single PE and the 8 PE configurations to keep the data in cache, therefore the Mflops rate is essentially the same and the speedup is as expected less than the number of PEs.

It follows that a model of parallel execution should assume that the instruction execution rate, I, is also a function of P and that

$$W(P) = P \times I(P) \times T(P).$$

3.4 The effects of blocking and idle threads of control

Blocking occurs when a thread of control wastes its cycles waiting for data produced by another thread. If $T_{blk}(P)$ denotes the blocking time and $T_{calc}(P)$ the computing time with P processing elements, then

$$T(P) = T_{calc}(P) + T_{blk}(P)$$

and

$$W(P) = P \times I(P) \times T_{calc}(P).$$

Empirical evidence for several problems examined in [9] suggests that the blocking time can often be modeled as linear in the number of processors. That is, there is a constant d (problem dependent) so that $T_{blk}(P) = d \times P$.

3.5 The model

We present a parallel computation model which takes into account all the factors described in the previous sections. The model assumes that $P_{min} \leq P \leq P_{max}$ and that the instruction execution rate $I(P)$ is constant over that range. The total work required by a computation with P threads of control is

$$W(P) = W(1) + W_{sd}(P) + W_{cc}(P)$$

and

$$W(P) = T_{calc} \times I(P) \times P$$

with $T(P) = T_{calc}(P) + T_{blk}(P)$ and $P > 1$. Then

$$S(P) = \frac{P}{PT_{blk}\frac{I(1)}{W(1)} + \frac{I(1)}{I(P)}\left(1 + f \times (P-1) + \frac{\bar{\theta}E}{W(1)}\right)} .$$

Two cases are now considered.

Case 1: $E = kP$. We have

$$S(P) = \frac{P}{aP+b} \quad \text{and} \quad \eta(P) = \frac{1}{aP+b}$$

with

$$a = \frac{I(1)}{I(P)}\left[T_{blk}\frac{I(P)}{W(1)} + f + \alpha\right], \quad b = \frac{I(1)}{I(P)}(1-f) \quad \text{and} \quad \alpha = \frac{\bar{\theta}k}{W(1)} .$$

The asymptotic speedup is easily calculated to be

$$S_\infty = \frac{1}{a}.$$

Case 2: $E = kP^2$. We have

$$S(P) = \frac{P}{aP^2+bP+c} \quad \text{and} \quad \eta(P) = \frac{1}{aP^2+bP+c}$$

with

$$a = \frac{I(1)}{I(P)}\alpha \quad b = \frac{I(1)}{I(P)}\left(T_{blk}\frac{I(P)}{W(1)} + f\right) \quad c = \frac{I(1)}{I(P)}(1-f).$$

A calculation shows that the maximum speedup is obtained with P_{opt} processing elements

$$P_{opt} = \sqrt{\frac{c}{a}} = \sqrt{\frac{1-f}{\alpha}}.$$

The maximum speedup is

$$S_{max} = S(P_{opt}) = \frac{1}{b + 2\sqrt{ac}} = \frac{1}{\frac{I(1)}{I(P)}\left[2\sqrt{\alpha(1-f)} + f + T_{blk}\frac{I(P)}{W(1)}\right]}.$$

The shape of the speedup curves when all the causes of inefficiency in a parallel computation are considered, are similar with the corresponding speedup curves in Figure 1.

Consider now the case when the blocking time is a linear function of the number of PEs, we have

$$T_{blk}(P) = d \times P \quad \text{for} \quad P > 1$$

and

$$S(P) = \frac{P}{P^2 d\frac{I(1)}{W(1)} + \frac{I(1)}{I(P)}\left[1 + f \times (P-1) + \frac{\tilde{\theta}E}{W(1)}\right]}.$$

If $E = kP$, then we have

$$S(P) = \frac{P}{aP^2 + bP + c} \quad \text{for} \quad P > 1$$

with

$$\begin{aligned} a &= d\frac{I(1)}{W(1)} \\ b &= \frac{I(1)}{I(P)}\left(f + \frac{k\tilde{\theta}}{W(1)}\right) \\ c &= \frac{I(1)}{I(P)}(1-f). \end{aligned}$$

The maximum speedup is now

$$S_{max} = \frac{1}{\frac{I(1)}{I(P)}[f + k\tilde{\theta}/W(1) + 2\sqrt{dI(P)(1-f)}]}$$

with

$$P_{opt} = \sqrt{\frac{W(1)(1-f)}{dI(P_{opt})}}$$

When $E = kP^2$ we have

$$S(P) = \frac{P}{aP^2 + bP + c}, \quad \text{for} \quad P > 1$$

with

$$\begin{aligned} a &= \frac{I(1)}{I(P)} \frac{k\bar{\theta}}{W(1)} + d\frac{I(1)}{W(1)} \\ b &= \frac{I(1)}{I(P)} f \\ c &= \frac{I(1)}{I(P)}(1-f). \end{aligned}$$

Thus when $E = kP^2$ and $T_{blk} = d \times P$ the optimum number of processors is

$$P_{opt} = \sqrt{\frac{W(1)(1-f)}{\bar{\theta}k + dI(P_{opt})}}$$

which gives a maximum speedup of

$$S_{\max} = \frac{1}{\frac{I(1)}{I(P_{opt})}[f + 2\sqrt{\frac{(1-f)}{W(1)} k\bar{\theta} + dI(P_{opt})}]} .$$

Note that in both these cases that, in order to know P_{opt}, one must solve a nonlinear equation of the form $P^2(C_1 + I(P)) = C_2$ with the known constants C_1 and C_2. When the effect of the communication complexity and the blocking are ignored, (when $k = 0$ and $d = 0$) and when $I(P) = I(1)$, then the previous expression gives the upper bound predicted by Amdahl's law.

The ratio $infl = \frac{I(P)}{I(1)}$ is called the *speedup inflation rate* and the value $S^{ef}(P) = S/(P)infl$ is called the *effective speedup*. When $E = \mathcal{O}(P^2)$ and $T_{blk} = \mathcal{O}(P)$ we have

$$S^{ef}_{max} = \frac{1}{f + 2\sqrt{\frac{(1-f)}{W(1)}(k\bar{\theta} + dI)}}$$

This upper bound of the effective speedup exhibits the behavior intuitively expected, namely

- it increases when the problem size increases (when $W(1)$ increases),
- it decreases when there is a significant amount of blocking (for example, when the computation is heavily imbalanced), and the instruction execution rate increases.

If the amount of duplicated and sequential work is insignificant ($f << 1$), then

$$S_{max}^{ef} \simeq \frac{1}{2}\sqrt{\frac{W(1)}{(k\theta + dI)}}.$$

In such cases the speedup is considerably lower than the upper bound predicted by Amdahl's law, as one would expect. Assuming that $infl(P) = \frac{I(P)}{I(1)} =$ constant for $P \geq Q$, it follows that when $E = kP^2$ and $T_{blk} = dP$, the relative speedup is

$$S_{P,Q} = \frac{S(P+Q)}{S(Q)} = \left(1 + \frac{P}{Q}\right) \frac{1}{1 + P\frac{aP+b+2aQ}{aQ^2+bQ+c}}$$

with a, b and c previously defined.

4 An In-Core 3-D FFT Algorithm for a Hypercube

4.1 The Algorithm

3-D FFTs are used in a variety of applications in science and engineering. For example, processing of seismic data, material sciences, and processing of X-ray diffraction data [10].

One approach for a 3-D FFT computation with $N = n_x \times n_y \times n_z$ data is to perform 2-D FFTs for, say, the $n_y(x \times z)$ planes and then to perform an additional 1-D FFT along the y axis. Clearly the other two orientations for the planes are possible. In turn, each 2-D FFT in the $(x \times z)$ plane could be performed as a sequence of, say, n_z 1-D FFTs along the x axis. With this view of a 3-D FFT, the obvious data partitioning strategy of an in-core computation is to assign to each PE one or more planes for the 2-D FFT transformation. Such a group of planes is called a *slab*. A slab orthogonal to the y axis is called an *y-slab* and one orthogonal to the z axis is called a *z-slab*. Figure 2 illustrates this data partitioning and shows that each PE needs to gather the *slices* of a slab assigned to all other PEs for the second step of the algorithm.

4.2 The Implementation

The kernel of a program [6] which implements these ideas, is presented in Figure 3. The node program uses two system calls, **numnodes** and **mynode** to get the size of the partition and the id of the current PE. Then the number and the orientation of planes in a slab, the slab width, d_y and d_z, and size, sb_z for y-slabs and z-slabs, and the slice size, sl_y are

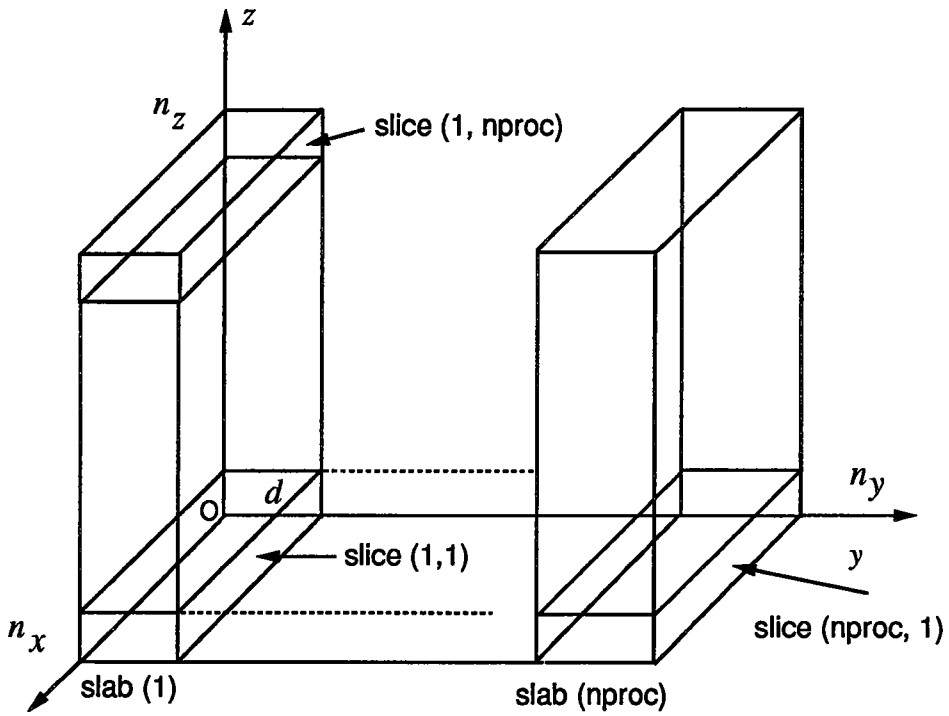

Figure 2: The slabs and slices for a data mapping with slabs orthogonal to the y axis.

determined. The PE performs a 2-D FFT on the d_y planes of the y-slab assigned to it. The second `for`-loop distributes slices of the y-slab processed by the PE to all other PEs, and gathers the slices of the z-slab to be processed by the PE.

To reduce the communication costs, *forced type messages* are used on the Intel iPSC/860. Such messages bypass the standard flow control mechanism and are stored directly into the user's buffer. But such messages are lost if a `receive` is not posted by the time the message arrives. For this reason, each node posts an asynchronous `receive` and then chooses a partner with whom it first synchronizes and to which it then sends the corresponding slice. The two `csend` statements seen in Figure 3 with zero length allow the current node `me` and its partner `my_partner` to synchronize. At each iteration of the second `for` loop, a pair (`me`, `my_partner`) is selected by both nodes. Indeed given a set of p integers from 0 to $p-1$ for any 3 integers i, j, k in this set, the following property holds

$$\text{if } j = \text{ xor } (i,k) \text{ then } i = \text{ xor } (j,k)$$

The communication complexity of this algorithm is $E = \mathcal{O}(P^2)$. Interestingly enough,

the communication is done in parallel by pairs of nodes and the communication pattern avoids link contention on a hypercube.

4.3 An Algorithm with $E = \mathcal{O}(P)$

An algorithm with $E = \mathcal{O}(P)$ is now outlined. Consider a minimum spanning tree like the one used for the broadcast/collapse mechanism in [5]. In the first phase of the algorithm, each node receives the y-slab(s) processed by its children, creates a super-slab and sends it to its parent. This phase terminates when the root node receives the entire 3-D mesh. Then the root performs the transposition of the mesh, creates super-z-slabs and sends them to its children. In turn the children distribute super-z-slabs and the process terminates when the leaves receive their z-slabs. It is very likely that the blocking time of the $\mathcal{O}(P)$ algorithm will be larger than the one for the $\mathcal{O}(P^2)$ algorithm.

4.4 Measurements

An experiment performed on an INTEL iPSC/860 consists of running a problem of fixed size (N given), but with different shapes of the mesh. The results of running the problem on a single node are reported in Table 2 and show that even though the amount of work, $W(1) = k' \times N \times \log N$ is the same, the i860 processor runs at different rates ranging from 7.5 to 17 Mflops. A plausible explanation in tune with results reported elsewhere [3], is that the i860 processor is very sensitive to the cache management. When the mesh is highly asymmetric ($16 \times 16 \times 1024$), the cache management is poor and the processor runs at a low rate, only 7.5 Mflops, but in the symmetric case ($64 \times 64 \times 64$ mesh), the cache management works well and the Mflops rate increases by a factor of almost 2.5.

Therefore, it is very plausible for many computations that each PE will run at one rate (probably higher) when the problem is solved with P processing elements and data is distributed across P processors, and at a different rate, possibly a lower one, when using only one processing element and all data is stored in one node only so that the cache management is likely to be poor. If this is true, then a problem which exhibits a low Mflops rate running in one node only will show an artificially high speedup with P processing elements. Conversely a problem which runs well (at a high Mflops rate) in one node will exhibit a smaller speedup with P processing elements, simply because it does not benefit from an increase in the Mflops rate per processor.

The results shown in Table 3 confirm this suspicion. Indeed, when running on a 16 node iPSC/860, the highest speedup is observed for the problem running at the lowest Mflops rate

```
    /* Find number of nodes in the machine partition */
nproc = numnodes ( )
    /* Get id of the current node */
me = mynode ( )
    /* Compute slab width, slab size, and slice size */
d_y = slab_width (ny, nproc)
d_z = slab_width (nz, nproc)
sb_y = slab_size (nx,nz,d_y)
sb_z = slab_size (nx,ny,d_z)
sl_y = slice_size (nx,d_y,d_z)
    /* Get the (me)-th y-slab and do a 2-D FFT */
my_slab = get_slab (me, slab_size)
for i=1 to d_y
     call fft2d(my_slab)
end_for
  /* Transposition loop. Node (me) has the (me)-th y-slab and needs
     (nproc-1) slices for the (me)-th z-slab from all other nodes. */
for other = 1 to nproc -1
    my_partner = xor (me, other)
    this_slice = mod (my_partner, nproc)
    msg_type = force_msg + other
      /*  Post asynch receive for a slice expected from my_partner */
    iget = irecv (msg_type, buffer, sl)
      /*  Send a message of zero length to my_partner and receives one
          from him to synchronize */
    call csend (other, dummy, 0, my_partner, procid)
    call crecv (other, dummy, 0)
    /* Send the slice my_partner needs */
    call csend (msg_type, this_slice, sl, my_partner, procid)
    /* Wait to get the slice I need from my_partner */
    call msgwait (iget)
    /* Save the data into node z-slab */
    copy (buf, my_slab(my_partner))
end_for
for i=1 to d
     call fftid (my_slab)
end_for
```

Figure 3: A program for 3-D FFT.

Mesh dimensions $n_x \times n_y \times n_z$	Instruction execution rate (Mflops)
$16 \times 16 \times 1024$	7.5
$16 \times 32 \times 512$	10.1
$32 \times 32 \times 256$	16.4
$64 \times 64 \times 64$	17.0

Table 2: The instruction execution rate of one node of the iPSC/860 running the 3-D FFT computation for a mesh of constant size, but with different shapes.

Mesh dimensions $n_x \times n_y \times n_z$	Instruction rate (Mflops)	Speedup	Instructions rate per processor
$16 \times 16 \times 1024$	74.29	9.90	4.64
$16 \times 32 \times 512$	93.49	9.25	5.84
$32 \times 32 \times 256$	121.55	7.41	7.59
$64 \times 64 \times 64$	124.21	7.30	7.76

Table 3: The instruction execution rate and the speedup for a 16 node iPSC/860 running the 3-D FFT computations for a mesh of constant size, but with different shapes. The instruction rate per processor is given, compare with Table 1.

on 16 nodes and on one node.

5 A Chebychev Iterative Algorithm for Solving a Linear System of Equations

Iterative methods are frequently used in different areas of scientific computing. For example, the phase refinement used in X-ray crystallography [10] is based upon an iterative scheme. Such methods are used in linear algebra for solving systems of linear equations. To guarantee numerical convergence iterative methods often require global synchronization after each iteration. Excessive communication and blocking are limiting the performance of such

methods on distributed memory MIMD systems [9]. Schemes which require less frequent synchronization are discussed in [7].

An experiment is described in this section which monitors the execution of the code implementing a Chebychev iterative algorithm for solving a linear system of equations, an important component of a parallel PDE solver. To ensure a load balanced execution, the domain decomposer, part of the //ELLPACK environment [4], attempts to assign to every PE an equal amount of computation. A careful selection of the interface points of the neighboring domains is also necessary in order to minimize and balance the communication cost. The experiment was conducted by taking two problems of a fixed size and repeating the execution with a number of PEs ranging from 2 to 128 for a rectangular domain and a 50×50 grid, and 4 to 64 PEs for an irregular domain and a 33×33 grid. An Ncube 1 was used.

The detailed behavior of all threads of control was captured by recording all the events, marking changes of state for every thread. For every event the TRIPLEX [5] tool creates a trace record, which contains the pertinent information about the event, type, time stamp, PE, amount of data transferred, etc. All the measurements reported are based upon a clock with resolution of 0.1 msec. To minimize the volume of trace data, only 5 iteration steps of the JACOBI SI method were performed and only events related to communication and control were recorded. Even so, the trace data collected during a single experiment with 128 PEs amounted to about 25 Mbytes.

For each case, the PDE solver executes using a variable number of PEs, ranging from 2 to 128 for the first case and 4 to 64 for the second case. The actual decomposition corresponding to the 64 PE case use 33×33 and 50×50 grid respectively. In the second case, the size of each subdomain and consequently the amount of computations performed by each PE is larger than for the first case. The number of interface points of each subdomain and consequently the amount of communication is also larger in the second case. For all the graphs in this section, the data associated with the first problem is presented with *solid* lines and +, while for the second one *dashed* lines and "x" are used. The two cases examined here show results in almost perfect qualitative agreement with one another.

Figure 4 presents the number of events per thread of control as a function of the number of threads on a logarithmic scale. This function provides a signature of the algorithm and its implementation. The algorithm requires global synchronization and at the end of each iteration each subdomain assigned to a PE needs to exchange boundary values with its neighboring subdomains. Communication was implemented by broadcasting and thus $E = \mathcal{O}(P^2)$.

The implementation uses broadcasting rather than multicasting for two reasons. First, the particular machine the experiment was carried on does not support efficient multicasting.

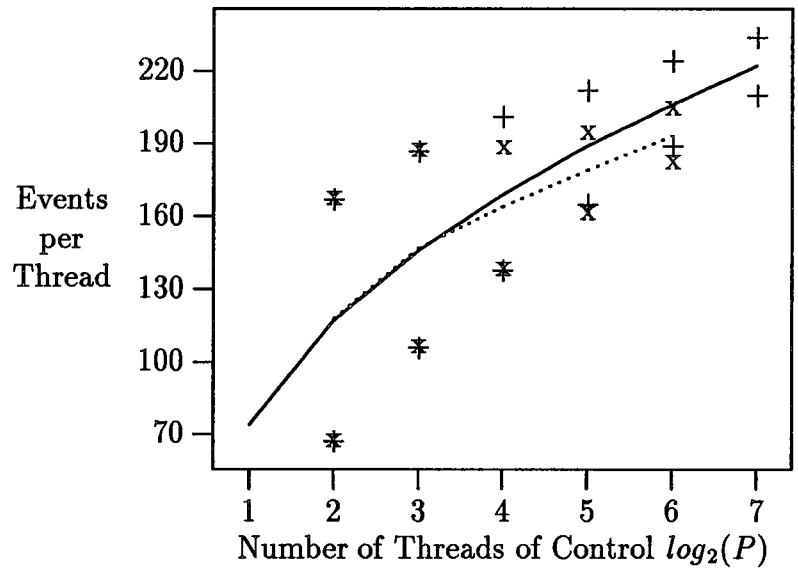

Figure 4: The expected number of *events* per thread of control and a 95% confidence interval for it.

Second, to multicast, each subdomain needs to know the ids of the PE where its neighboring subdomains are assigned. An algorithm which statically assigns subdomains to PEs so as to preserve geometric proximity was not available, due to the complexity of the subdomain shapes. Thus, efficient multicasting was not possible even if it were available. Any algorithm to map dynamically logical subdomain ids to physical processors requires additional communication.

Some events lead to blocking phenomena. This is the case of the synchronous read operations when the PE executing the thread of control blocks waiting for data to become available.

The blocking time as a fraction of the read time is shown in Figure 5, it increases from about 70% in the case of four threads to more than 90% in the case of 64 threads.

Since blocking is an important source of low processor utilization and consequently of low speedup, the blocking phenomena deserves to be further scrutinized. In Figure 6, a possible scenario involving communication between two PEs, PE_i and PE_j is shown. At time t_1, PE_i issues a READ requesting data from PE_j. At time t_2, PE_j has the data available and issues a WRITE to PE_i. The interval $\tau_1 = t_2 - t_1$ is called *algorithmic blocking time.* Figure 6 shows that the communication time $\tau = t_4 - t_1$ has three components, algorithmic

blocking, propagation (τ_2), and data transmission time (τ_3), so we have $\tau = \tau_1 + \tau_2 + \tau_3$. The algorithmic blocking occurs when the need for data at the consumer's site precedes the actual generation of data at the producer's site. The propagation delay measures the time it takes for one bit of information to travel from the source to the destination. It depends upon the interconnection network and upon the routing method used, it may include some time needed to establish the connection. The data transmission time measures the time to send the data and it depends upon the amount of data being transmitted and upon the hardware speed.

Table 4 shows the expected algorithmic blocking time per read operation. From Table 4 and Figure 5, it follows that most of the blocking is in fact algorithmic blocking, therefore increasing the communication speed is not likely to have a significant effect upon this computation.

Numer of threads of control ($\log_2 P$)	1	2	3	4	5	6	7
Average Duration of a Read (ticks)	36	91	111	121	162	195	226
Average Blocking time per Read (ticks)	11	46.8	63	86	118	158	188
Algorithmic blocking as fraction of the read time (%)	30	51	56	71	72	86	83

Table 4. Read time (ticks) and algorithmic blocking time.

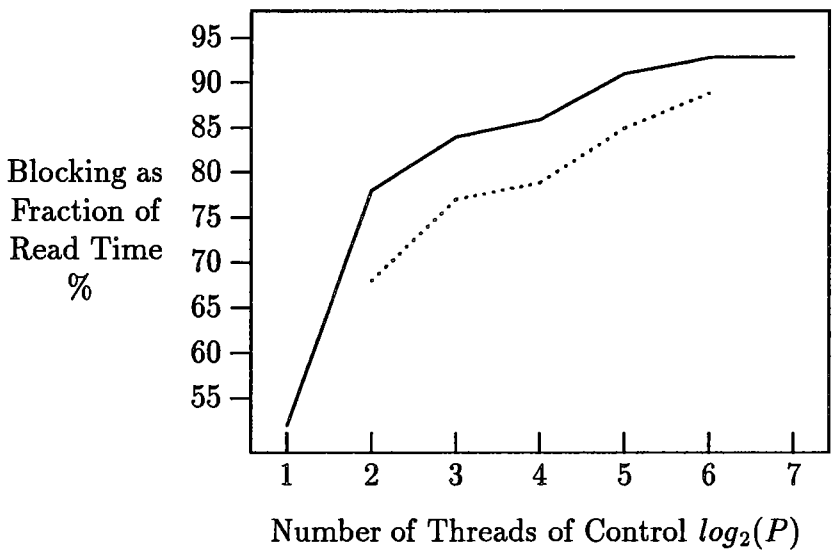

Figure 5: The expected *blocking time* during a read operation as fraction of the total read time during a read operation.

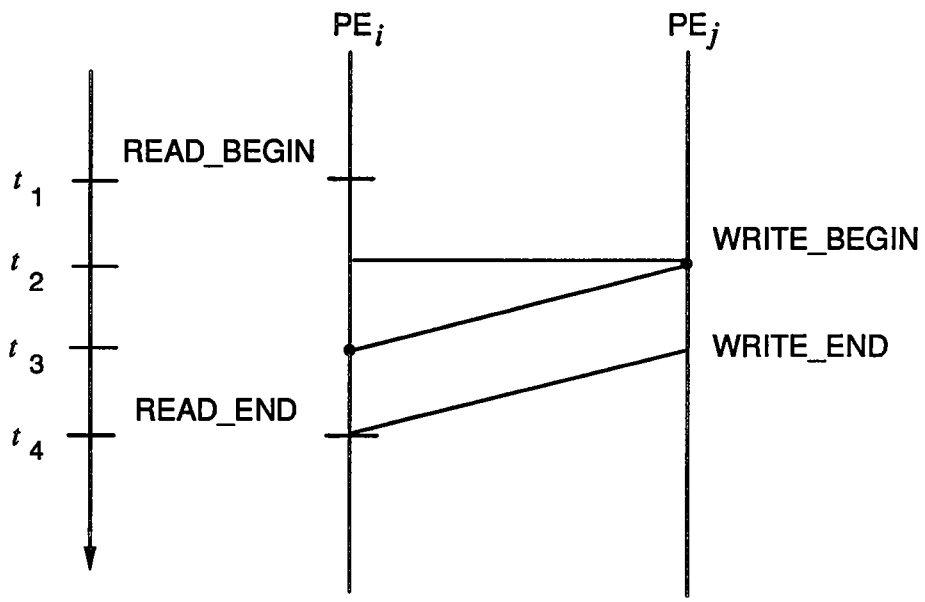

Figure 6: Communication involving blocking.

6 Conclusions

This paper discusses the use of the speedup as a measure of performance of a parallel computer. It points out that execution time and the space requirements of some of the applications running on distributed memory MIMD systems like the Touchstone Delta system are so large that speedup cannot be determined simply because the sequential computation requires a prohibitively large amount of time. It also shows that the processor architecture, in particular the amount of cache available can distort the results of simple minded measurements. Superlinear and/or very large speedups may be due to an inefficient sequential computation rather than a very efficient parallel one. One of several causes of inefficiency of the sequential computation is poor cache management due to a large data space.

To preserve the virtues of a dimensionless measure of performance we propose the relative speedup which compares the execution time over a range of processing elements. This range should be determined so that the execution is carried out in comparable terms, e.g., similar instruction execution rates, measurable execution time and so on.

A model of a parallel execution which takes into account the major causes of inefficiency in a parallel computation, the sequential and duplicated work, communication and control as well as blocking is presented. This model allows the characterization of a parallel algorithm independent of the architecture. For example, an algorithm with communication complexity $E = \mathcal{O}(P)$ is more scalable than one with $E = \mathcal{O}(P^2)$ on any architecture. This means that the maximum speedup attainable is higher and can be obtained with a larger number of PEs as shown in §3.2. Yet even this high level characterization can be misleading. For example the 3D FFT algorithm discussed in §4 has a communication complexity $E = \mathcal{O}(P^2)$ but allows parallel communication. The set of PEs is partitioned into groups of size 2, all groups communicate in parallel and there is no contention for communication channels. Such an algorithm may run more efficiently than an $\mathcal{O}(P)$ algorithm in which communication is strictly sequential.

The blocking model of a parallel algorithm is very important. An asynchronous algorithm is more efficient than one which requires global synchronization. An algorithm in which the blocking time $T_{blk} = \mathcal{O}(P)$ is less efficient than one with $T_{blk} = \mathcal{O}(\log \mathrm{P})$.

Last but not least the life time of intermediate results has a significant impact upon the performance of the parallel algorithm.

The design of a parallel algorithm is considerably more complex than that of a sequential one. Data partitioning, mapping, control and load balancing mechanisms are an integral part of any control parallel algorithm.

Multiple tradeoffs are involved. Communication, space, time, and control complexities have to be balanced in an optimal way to achieve the best performance on a given architec-

ture. For example, the communication complexity can be reduced by increasing the space requirements as shown in §4.3. The communication complexity can be reduced by increasing the amount of the duplicated work. The life time of intermediate results can be reduced by increasing the blocking time as the scheme discussed in §4.2 shows. The scalability of an algorithm can be traded off for schemes which allow parallel and contention free communication.

Acknowledgments

The authors express their thanks to Dr. Robert Lynch for suggesting some of the experiments concerning the 3-D FFT, to Dr. E. Vavalis for his contribution to the measurements reported in §5, to M. Cornea-Hasegan for carrying out the measurements reported in §3.3.

Literature

1. Amdahl, G., "The validity of a single processor approach to achieving large scale computing capabilities", in *AFIPS Proc.*, **30** (1967), pp. 483–485.

2. Gustafson, J.L., Montry, G.R., and Benner, R.E., "Development of parallel methods for a 1024-processor hypercube", *SIAM J. Sci. Statist. Comput.*, **9** (1988), pp. 609–638.

3. Heath, M.T., Geist, G.A., and Drake, J.B., "Early experience with the Intel iPSC/860 at the Oak Ridge National Laboratory", *ORNL/TM-11655* (1990).

4. Houstis, E.N., Papatheodorou, T.S., and Rice, J.R., "Parallel ELLPACK: An expert system for parallel processing of partial differential equations", *Math. Comp. Simulation*, **5** (1990), pp. 63–73.

5. Krumme, D.W., Couch, A.L., and House, B.L., "The TRIPLEX tool set for the NCUBE multiprocessors", Technical Report, Tufts University, (1989).

6. Kushner, E., *In core 3-D FFT program for iPSC/860*, Private communication. Also Intel, iPSC/860 Basic Math Library User's Guide (1991).

7. Marinescu, D.C., and Rice, J.R., "Synchronization and load imbalance effects in distributed memory multiprocessor systems", *Concurrency: Practice and Experience*, **3**, (1991), pp. 593–625.

8. Marinescu, D.C., and Rice, J.R., "On high level characterization of parallelism", *Journal of Parallel and Distributed Computing*, (1992), to appear.

9. Marinescu, D.C., Rice, J.R., and Vavalis, E., "Performance of iterative methods on distributed memory processors", *Applied Numerical Methods*, (1993), to appear.

10. Marinescu, D.C., Rice, J.R., Cornea-Hasegan, M., Lynch, R.E., and Rossmann, M.G., "Macromolecular electron density averaging on distributed memory MIMD systems", CSD-TR 92-019, (1992).

11. Worley, P.H., "The effect of time constraints on scaled speedup", *SIAM J. Sci. Stat. Comp.*, **11** (1990), pp. 838–858.

12. *Proc. First Intel Delta Workshop*, Messina P. and Mihaly T., Eds. (1992).