Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1990

Synchronization and Load Imbalance Effects in Distributed Memory Multiprocessor Systems

Dan C. Marinescu

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

Report Number:
90-1000

Marinescu, Dan C. and Rice, John R., "Synchronization and Load Imbalance Effects in Distributed Memory Multiprocessor Systems" (1990). *Department of Computer Science Technical Reports.* Paper 3.
https://docs.lib.purdue.edu/cstech/3

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

To appear in Concurrency Practice and Experience, 1991.

SYNCHRONIZATION AND LOAD IMBALANCE EFFECTS IN DISTRIBUTED MEMORY MULTIPROCESSOR SYSTEMS*

Dan C. Marinescu
and
John R. Rice

Computer Sciences Department
Purdue University
Technical Report CSD-TR-1000
CAPO Report CER-90-29
Revised June 1991

* Work supported in part by the Strategic Defense Initiation through ARO grants DAAG03-86-K-0106, DAAL03-90-0107 and by National Science Foundation grant CCR-8619817.

To appear in Concurrency Practice and Experience, 1991.

Synchronization and Load Imbalance Effects In Distributed Memory Multiprocessor Systems

Dan C. Marinescu, John R. Rice
Computer Science Department
Purdue University
West Lafayette, IN 47907, U.S.A.

June 24, 1991

Abstract

Synchronization is a major cause of wasted computing cycles and of diminished performance in parallel computing. This paper investigates the effects of synchronization upon the performance of iterative methods on distributed memory MIMD machines. A quantitative analysis of the effects of the communication latency and of the load imbalance due to the nondeterministic execution times for iterative methods is presented. This analysis explains the rather poor performance observed often in actual implementations of such methods and suggests better ways to achieve convergence without frequent synchronization.

1 Introduction

Numerous parallel algorithms and methods require some form of *synchronization*, or coordination of concurrent activities. To synchronize, a processor executing a thread of control may block waiting for the data produced by another thread, or for an external event to occur. There are cases when all the threads of control have to reach a consensus before proceeding to the next step and *global synchronization* is necessary. For example, iterative methods require a processor to exchange boundary values with its neighbors, but only after *all of them* have finished the current iteration.

Synchronization is a major cause of wasted computing cycles and of diminished performance in parallel computing. Synchronization is analogous to stopping at a traffic light, the more red lights that are encountered on a given route, the longer is the time to reach the destination. Avoiding synchronization is equivalent to taking a highway instead of a city street. The method of self-synchronization discussed in Section 3 amounts to traveling on a boulevard at a speed dictated by the "green wave" to avoid the penalty of red lights.

This paper investigates the effects of synchronization upon the performance of iterative methods on distributed memory MIMD machines. The discussion is restricted to the so-called SPMD execution, when the same program is executed by all the processors of a parallel system on different data sub-domains. The SPMD paradigm is widely used because divide and conquer methods are very popular for solving large numerical problems. Moreover, it is fairly difficult to write and debug a large number of different programs, one for each individual computation executing on each of the processing elements of a MIMD machine.

On a distributed memory system the synchronization is done by exchanging messages and its performance depends upon the communication latency. The communication on such a machine is fairly expensive, currently, the time to exchange a short message between two nearest neighbors

is typically equivalent to the time to execute a few hundred floating point operations. As new generations of distributed memory MIMD machines emerge, the processor bandwidth (the instruction execution rate) increases at a faster rate than the communication speed, therefore the effects discussed in this paper become more and more severe. A possible solution for future machines is to have a a dedicated, very fast communication network for short messages needed for synchronization.

Most iterative methods require global synchronization, an operation which involves the exchange of a large number of messages and it is a source of considerable inefficiency on distributed memory MIMD machines. Yet, explicit synchronization could be avoided if two conditions are met. The computations have to start at the same time and their duration must be the same. But even in case of SPMD execution the data dependency makes this approach unfeasible. Different processors reach the synchronization point at different times even in the load balanced case because they execute different instructions streams and even the execution time of an instruction is data dependent.

This paper is devoted to a quantitative analysis of the effects of the communication latency and of the load imbalance due to the nondeterministic execution times for iterative methods. This analysis explains the rather poor performance observed often in actual implementations of such methods and suggests better ways to achieve convergence without frequent synchronization.

The paper is focused upon distributed memory MIMD systems because they are emerging as a viable alternative to other computer architectures for solving very large problems in areas like signal processing, structural analysis, fluid mechanics, molecular biology, etc. [2,3]. As an example of a very large problem consider the molecular replacement real space averaging method described in [17]. To produce an electron density map the method needs considerable computing resources, tens of hours of CPU time of Cyber 205 and enough memory for 3.24×10^9 grid points for a particular algorithm and unit cell dimensions. Dozens of such maps must be computed to determine the structure of one virus.

Distributed memory MIMD systems with hundreds of *processing elements*, PEs, able to deliver tens of GFLops and with Gbytes of memory have been built. The INTEL Touchstone Delta machine of the Concurrent Supercomputer Consortium is a 2-D mesh with 520 numeric nodes, each with 16 Mbytes of memory and with each processor capable of delivering tens of Mflops. If the current trend continues, it is reasonable to expect that in the mid 90's machines will be available with thousands of PEs able to deliver tens of Tflops and with tens of Tbytes.

Though other types of interconnection networks like multi-stage networks, rings or meshes are used in MIMD systems, the hypercubes are the most common ones. In a hypercube of order L, there are $N = 2^L$ PEs, located in the nodes of a cube in the space of L dimensions. Each node is connected to exactly L other nodes and the maximum distance between any pair of nodes, the number of links to be traversed from one node to the other, is at most L. The overall communication bandwidth of a hypercube increases with its size, since the communication bandwidth is proportioned with the number of links connecting each node with the rest of the cube. When the size of the cube increases by one, from order L to order $L+1$, the number of processing elements doubles, but the number of links emerging from each node increases only one one. This makes the hypercube machine attractive to build and well balanced in terms of the computation to communication bandwidth ratio. The cost performance ratio of such machines makes them very attractive for many applications. These factors explain the popularity of the hypercubes. Numerous hypercubes have been built and the work reported in this paper was carried out using one of the commercially available machines, an NCUBE/1. Impressive results have been reported using such machines [3], but not all types of parallel numerical computations are suitable for implementations on hypercubes, primarily due to the high cost of communication.

This paper is organized as follows: A fluid approximation for the analysis of data dependencies upon load balance and models for several synchronization structures are discussed in Section 2.

Section 3 discusses the effects of communication latency upon synchronization, and finally Section 4 presents performance measurements.

2 Load Imbalance Effects in SPMD computations

The speed up is defined as the ratio of the execution time with one processor to the execution time with P processors and is a measure of the performance of a parallel computation. To obtain a large speed up, it is necessary to maintain a high processor utilization, the time the processors are idle needs to be kept as low as possible. To do so, the load assigned to different processors must be balanced. *A static load balancing* in the SPMD execution means to assign to every PE balanced or equal data subdomains with the hope that during execution, different processors will have dynamic loads close to one another. But as pointed out earlier static load balancing does not provide a guarantee of *dynamic load balancing.* Due to data dependencies, the actual flow of control of different PEs is different, different PEs execute different sequences of instructions and they arrive at a synchronization point at different times. Hence starting computations at the same time on all PEs does not guarantee that they will terminate at the same time even when their loads are statically balanced. A first objective of the paper is to model the effects of data dependencies upon the load balance and to report upon the actual measurements of the execution times on a particular application. If the study of data dependencies allows us to compute the load imbalance factor Δ, (defined in Section 2.1), then methods to reduce the effects of communication latency upon synchronization can be considered. For example, a method of self synchronization discussed in Section 3.5 proposes that every PE enters the communication period at the end of each iteration only after a time equal to $\mu(1+\Delta)$ with μ the expected execution time per iteration. Then, an explicit synchronization takes place only every Rth iteration.

From this brief presentation it follows that even if the effects of data dependencies are relatively minor, so they may increase the actual execution time by a few percent, the analysis of these effects is important in order to design schemes which prevent the frequent need for synchronization by requiring processors to enter periods of *self blocking.*

2.1 Load balance and data dependency in SPMD execution

Consider a parallel computation C and a multiprocessor system with P identical PEs, π_0, π_1, ..., π_{P-1}. It is assumed that the same program is executed by every PE using different data and that C requires global synchronization. In other words, C consists of say n subcomputations C_i, $1 \leq i \leq n$ such that any PE, π_0 starts C_{i+1}, only after all PEs have completed C_i. The period when C_i is executing is called the *i-th epoch* $\mathcal{E}_i$ of the computation.

For simplicity assume first that communication occurs instantaneously and attempt to quantify the effects of data dependencies upon the traditional measures of efficiency in parallel processing, namely, the processor utilization U, or its dual, the speed up with P processors

$$S_P = PU. \tag{2.1}$$

Since $U \leq 1$ we have $S_P \leq P$.

To achieve an algorithmic load balance, it is customary to perform an *equipartition* of the data domain, namely, to assign to every PE, π_j a data subdomain $\mathcal{D}_j$ of equal size. First assume that all P data subdomains are identical, they contain the *same* data. If E_i is the time required by an PE to perform C_i, then in absence of any delay due to hardware failures, all PEs are expected to complete C_i, $1 \leq i \leq n$ at the same time, since they have started at the same time. The time

required by C_i is denoted as T^i and it is called the *duration of the i-th synchronization epoch.* The previous arguments indicate that in case of a strictly deterministic execution time, all PEs can be kept running at all times during the execution of C and then $U = 1$.

When the P processors execute C_i using different data subdomains, the execution time X_j^i of each π_j will depend upon the pair $(\mathcal{E}_i, \mathcal{D}_j)$, it is data dependent. In general, each PE will execute a different sequence of instructions.

To model the nondeterminism due to data dependencies X_j^i, $1 \leq j \leq P$ will be considered independent identically distributed random variables with mean μ_i and variance σ_i for all synchronization epochs, $1 \leq i \leq n$. In this case the duration of the i-th synchronization epoch will be a new random variable

$$T^i = \max(X_1^i, \ldots, X_P^i). \tag{2.2}$$

Clearly, the average processor utilization will be $U < 1$, since some PEs will finish execution of C_i before the others.

As shown in [8], the expected value of T^i can be expressed as

$$E(T^i) = \mu_i(1 + \Delta_i) \tag{2.3}$$

with

$$\Delta_i = f(P) \cdot g(C_X) \tag{2.4}$$

with $C_X = \frac{\mu_X}{\sigma_X}$ and f and g depend upon the actual distribution of X^i and upon the number P of PEs.

For a uniform distribution we have [8]

$$f(P) = \frac{P-1}{P+1} \tag{2.5}$$

$$g(C_X) = C_X\sqrt{3}. \tag{2.6}$$

For the exponential distribution, $g(C_X) \equiv 1$ and $f(P) = \log P + C$ with C is Euler's constant, $C = 0.577$. For the standard normal distribution $g(C_X) \equiv 1$ and

$$f(P) \approx (2\log P)^{1/2} - \frac{1}{2}(2\log P)^{-1/2}(\log\log P + \log 4\pi - 2C). \tag{2.7}$$

To analyze the effects of data dependencies we propose a *fluid approximation* in which we replace the stochastic duration of a synchronization epoch T^i by its average value $E(T^i)$. In this approximation, the average processor utilization is

$$U = \frac{\sum_{i=1}^{n} \mu_i}{\sum_{i=1}^{n} \mu_i(1 + \Delta_i)}. \tag{2.8}$$

It is more convenient to compute a load imbalance factor defined as

$$\psi = \frac{\sum_{i=1}^{n} \mu_i \Delta_i}{\sum_{i=1}^{n} \mu_i}. \tag{2.9}$$

Then the average processor utilization U is related to ψ by

$$U = \frac{1}{1 + \psi}. \tag{2.10}$$

The computation of ψ is slightly more difficult when the number of PEs in different synchronization epochs is different. The next two sections explore such situations.

2.2 Analysis of two synchronization structures

We discuss two synchronization structures which can be used in conjunction with domain decomposition techniques for solving partial differential equations (PDE's). We restrict our analysis to the effects of load imbalance. The two synchronization structures use at most $N = a^K$ processors. These structures are presented in Figure 1 for the particular case $a = 2$ and $K = 3$. The first structure (Figure 1a) is characterized by the following properties:

P1. The computation consists of $K + 1$ epochs and the number of active processors in the i-th epoch is $I_i = a^{K-i}$ with $a > 1$. In the first epoch there are $I_0 = N = a^K$ active processors. It follows that $\Delta_i > 0$ for $i > 0$ and $\Delta_K = 0$ since there is only one processor active during the last epoch.

P2. The execution time of all tasks in all synchronization epochs are independent, identically distributed random variables $X_{i,j}$ with mean μ_X, variance σ_X and coefficient of variation C_X.

P3. There is a global synchronization among all tasks of a given epoch.

The second structure (Figure 1b) is characterized by the following properties:

P1. The same number of active processors as the first structure.

P2. The same assumption on the execution times as for this structure.

P3. Within a given epoch, the tasks are synchronized in groups of a processors. There is no global synchronization between epochs and, as soon as a related set of tasks completes in epoch i, the descendent task commences in epoch $i + 1$.

We believe that the synchronization structures shown in Figure 1 are common among those that arise in parallel computation. We note some applications here to illustrate the variety that exists. We do not attempt to explain them in detail, as that detracts from the object of this paper.

A principal source is in models of physical phenomena (e.g., heat flows, electromagnetic forces, stresses, air flows) which are modeled by differential equations in 1 to 4 physical dimensions. These phenomena are inherently local in time and space. They are inherently synchronized in time, but loosely synchronized in space; space synchronization comes through the time for effects to propagate through space via local interactions.

Computations modeling these phenomena can exploit this loose coupling in space to achieve parallelism. The principal technique is called *domain decomposition*, where physical space is decomposed into a large number of domains. Since interactions between these domains is local, this allows one to use locally connected computer architectures effectively. See [9-11], [13] and [14] for previous work of ours, which include descriptions of this approach at a fairly high level. The first control structure arises in multigrid and other multilevel iterations for PDEs, the second arises in divide and conquer methods, e.g., nested dissection. There is an enormous literature [4], [16] on the mathematical analysis of specific instances of this general approach, this is currently one of the most intensively studied areas of numerical computation.

The basic technique is to compute the results in the interior of a particular domain and then communicate the new state to neighboring domains for their use. Important characteristics of such computation are as follows:

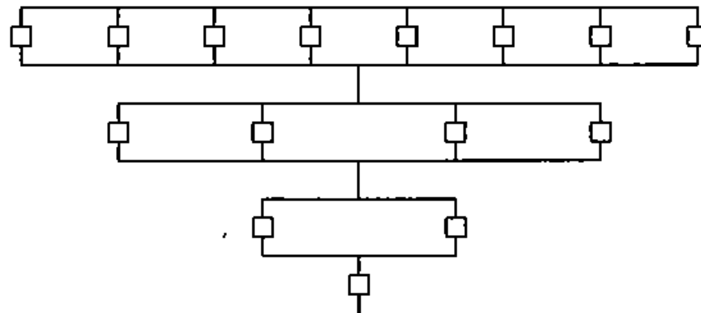


Figure 1a

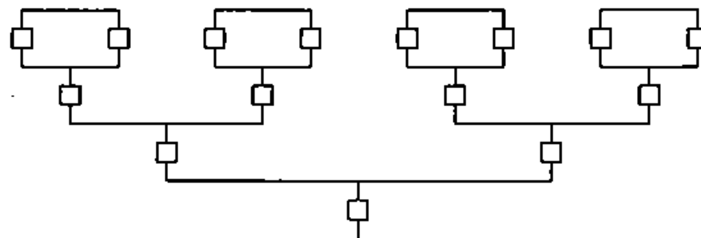


Figure 1b

FIGURE 1: The two synchronization structures for $a = 2$ and $K = 3$. The structures are (1a) an exponentially decreasing number of processors with rate a, (1b) a tree with a processors in parallel at each branch.

1. The interior computations and data are usually large compared to data to be communicated. One of the objectives of algorithm design is to be sure this is so, it follows naturally if one chooses domain shapes that have small "surface" compared to "volume" (e.g., nearly spheres or cubes).

2. The interior computations are usually similar (use the same program), but rarely identical (have different data) due to variations in shapes, materials, intensities of physical effects, etc.

3. Synchronization in time is essential. Some models of the physics may compensate for small time asynchronizations locally, and more as domains become separated in space. Many algorithms have an artificial time (e.g., iteration numbering) which has the same characteristics as real world time.

There are other applications such as graphics, image processing, searching unstructured data and text processing where parallelism is inherent in the nature of the problem. The structures considered all occur in specific examples of these applications.

The key ingredients in the analysis of these applications for parallel computations are:

* the ratio γ of computing X_j^i costs to communication costs,

* the ability to partition the problem so processors have equal expected computational loads (the X_j^i have the same mean μ_i),

* the distribution of the execution times X_j^i.

In many of the applications, it is practical to allocate the expected computational loads fairly equally. Each processor is assigned an equal volume of space, an equal area of a display or an equal amount of text to typeset. Of course, one can easily construct examples where this allocation is not easy, but even then, one can expect that an effective allocation is made. The distribution of the X_j^i can also vary greatly, but again we expect that many, if not most, applications will have the X_j^i distributed "tightly" about the mean μ_i. Thus a model assuming a uniform or normal distribution for the X_j^i will be appropriate for many applications.

In Table 1, we present a list of applications along with rough orders for the ratio of computation and communication costs. The entries are only orders of magnitude, numerical factors can vary greatly. The quantity S merely measures the size or "bulk" of the computation assigned to a processor, it is not the same from line to line, nor is it necessarily a variable that appears in an algorithm for these applications. We see that this ratio increases with problem sizes for all these applications, sometimes dramatically. In machines that can overlap communication with computing, the effective communication cost (the value of β) may be much less than the values indicated in Table 1.

The load imbalance costs for the two synchronization structures discussed previously are investigated. It is assumed that the execution times in all synchronization epochs have the same distribution F_X. The strategy is to compute the total load imbalance costs and then the load imbalance factor, ψ, for each structure and for several distributions of the execution times.

The First Synchronization Structure

Let us denote by $\Delta_{a,K}^{(1)}$, the total load imbalance for the first structure (Figure 1a)

$$\Delta_{a,K}^{(1)} = \sum_{i=0}^{K-1} \Delta_i \qquad (2.11)$$

The effects of the load imbalance are characterized by $\psi_{a,N}^{(1)}$ which is defined as the ratio of the expected increase of the execution time due to load imbalance, to the parallel execution time in absence of any load imbalance, that is

$$\psi_{a,K}^{(1)} = \frac{\mu_X \Delta_{a,K}^{(1)}}{\mu_X} \left(\frac{1}{(\log_a N + 1)} \right) = \frac{\Delta_{a,K}^{(1)}}{K+1}. \tag{2.12}$$

The case when $a = 2$ and $K = \log_2 N$ is of special interest. Then

$$\psi_{2,K}^{(1)} = \frac{\Delta_{2,K}^{(1)}}{K+1}. \tag{2.13}$$

Exact expressions for $\Delta_{a,K}^{(1)}$ can be obtained for the uniform, and the normal distributions of the execution time in each epoch. After presenting these results we give an upper bound for the case when the distribution F_X of X is not known.

TABLE 1: *Applications Using Hierarchical Synchronization Structures.* The order of magnitude of the computing/communication ratio is given as a function of the size of the computation assigned to a processor.

Application	Ratio of Computing to Communication	
	2 Dimensions	3 Dimensions
Domain decomposition for PDEs		
* using Gauss elimination		
(multifront methods)	$S^4/S = S^3$	$S^7/S^2 = S^5$
* using SOR iteration	$S^3/S = S^2$	$S^4/S^2 = S^2$
* using FFT method	$S^2/S = S$	$S^3/S^2 = S$
* multigrid iteration	$S^2/S = S$	$S^3/S^2 = S$
Time dependent PDE solutions		
* explicit methods	$S^2/S = S$	$S^3/S^2 = S$
* implicit methods		
Gauss elimination	$S^4/S = S^3$	$S^7/S^2 = S^5$
iteration	$S^2/S = S$	$S^3/S^2 = S$
* ADI methods	$S^2/S = S$	$S^3/S^2 = S$
Newton iteration for nonlinear PDE		
* Gauss elimination	$S^4/S = S^3$	$S^7/S^2 = S^5$
* iteration	$S^3/S = S^2$	$S^4/S^2 = S^2$
FFT methods	$S^2/S = S$	$S^3/S^2 = S$
Graphics		
* simple display	$S^2/0 = -$	$S^3/0 = -$
* connected displays (contours)	$S^2/S = S$	$S^3/S^2 = S$
* feature extraction	$(S^2 \to S^3)/S = S \to S^2$	$(S^3 \to S^4)/S^2 = S \to S^2$
Search in amorphous data	$S/1 = S$	
Text processing (Tex, troff)	$S/1 = S$	
Numerical integration	$S^2/1 = S^2$	$S^3/1 = S^3$
Discretization of PDEs	$S^2/S = S$	$S^3/S^2 = S$

The Uniform Distribution. In this case the ratio of the load imbalance costs to the parallel execution time is given by

$$\psi^{(1)}_{a,K} = C_X\sqrt{3}\,\frac{[K - 2\times Q_{a,K}]}{K+1} \tag{2.14}$$

with

$$Q_{a,K} = \sum_{i=0}^{K-1} \frac{1}{1+a^i}. \tag{2.15}$$

This expression is derived using the load imbalance cost for a synchronization epoch with a uniform distribution of the execution time

$$\Delta_i = C_X\sqrt{3}\,\frac{I_i - 1}{I_i + 1} \;=\; C_X\sqrt{3}\,\frac{a^{K-i} - 1}{a^{K-i} + 1}. \tag{2.16}$$

Then the total load imbalance costs are

$$\Delta^{(1)}_{a,K} = \sum_{i=0}^{K-1} \Delta_i = C_X\sqrt{3}[K - 2Q_{a,K}]. \tag{2.17}$$

Observe that

$$\sum_{i=0}^{\infty} \frac{1}{1+a}i \;=\; \frac{1}{2}\,\Phi_{21}\,(a, -1; -a; 1) \tag{2.18}$$

with Φ_{21} the basic generalized hypergeometric function. It seems nontrivial to derive exact expressions for $Q_{a,K}$ and we derive bounds for it. We see immediately that

$$\frac{A}{a} \le Q_{a,K} \le A. \tag{2.19}$$

with

$$A \;=\; \frac{a^K - 1}{a^{K-1}(a-1)} \;=\; \frac{a}{a-1}\left(1 - \frac{1}{N}\right) = \frac{a(N-1)}{N(a-1)}. \tag{2.20}$$

It follows that

$$C_X\sqrt{3}(K - 2A) \le \Delta^{(1)}_{a,K} \le C_X\sqrt{3}\left(K - 2\,\frac{A}{a}\right). \tag{2.21}$$

When $a = 2$ and N is large $A \approx 2$. In this case

$$C_X\sqrt{3}(K-4) \le \Delta^{(1)}_{2,K} \le C_X\sqrt{3}(K-2). \tag{2.22}$$

To conclude the discussion for the uniform distribution and the first synchronization structure we plot $\psi^{(1)}_{2,K}$ in Figure 2 for the binary case, $a = 2$, and for different values of C_X. For small values, say $C_X = 0.01$, the effect of load imbalance is hardly noticeable. For larger C_X, the load imbalance can add as much as 30% to the parallel execution time when N is large ($N > 2^{32}$).

The Normal Distribution. Consider first the case of a standard normal distribution. In Appendix 1 we show that

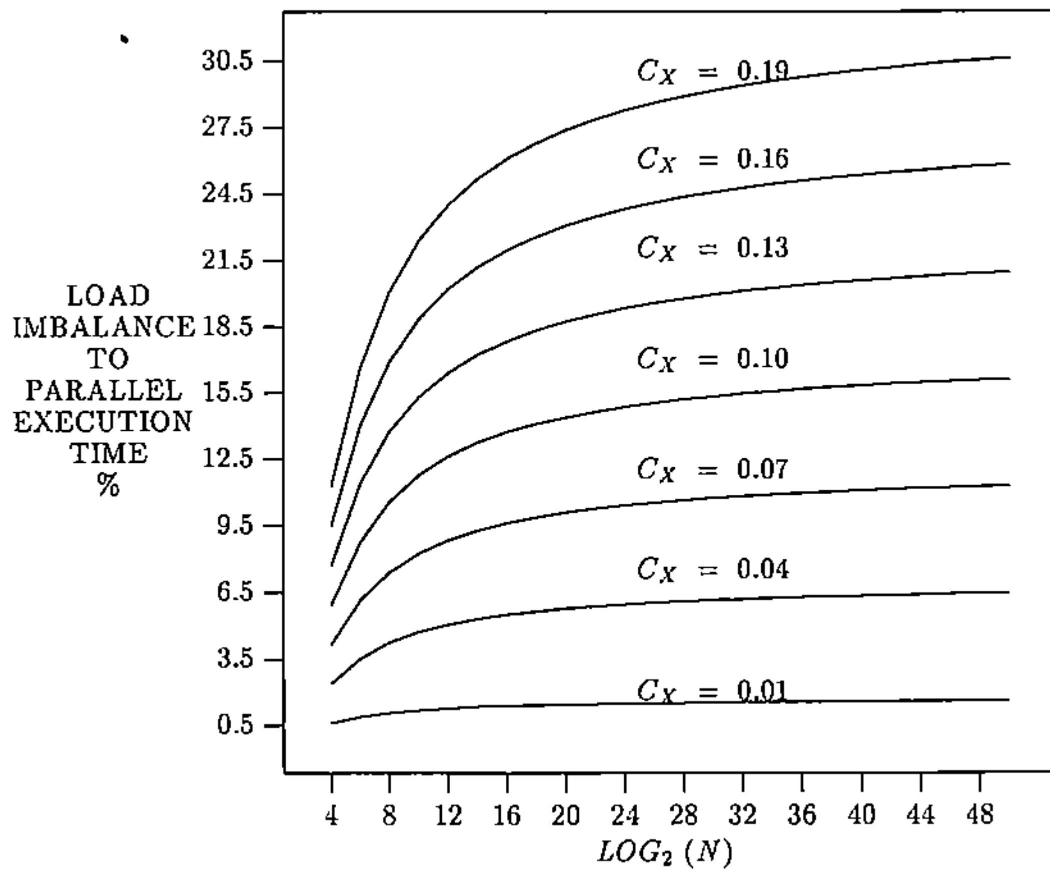


FIGURE 2: The ratio $\psi^{(1)}_{2,k}$ of load imbalance costs to the parallel execution time in absence of any load imbalance effects as function of the problem size for the first synchronization structure. The execution time has a uniform distribution with coefficient of variation C_X.

$$\psi^{(1)}_{a,K} = \frac{1}{K+1}\left(A(a)\cdot S_1(K) - B(a)\cdot S_2(K) - \frac{1}{2A(a)}\cdot S_3(K)\right) \tag{2.23}$$

with

$$A(a) = (2\log a)^{1/2} \tag{2.24}$$

$$B(a) = \frac{1}{2A(a)}\left[\log 4\pi - 2C + \log\log a\right] \tag{2.25}$$

$$S_1(K) = \sum_{i=1}^{K-1}(K-i)^{1/2} = \sum_{i=1}^{K-1}(i)^{1/2} \tag{2.26}$$

$$S_2(K) = \sum_{i=1}^{K-1}(K-i)^{-1/2} = \sum_{i=1}^{K-1}(i)^{-1/2} \tag{2.27}$$

$$S_3(K) = \sum_{i=1}^{K-1}(K-i)^{-1/2}\log(K-i) = \sum_{i=1}^{K-1} i^{-1/2}\log i. \tag{2.28}$$

When $a = 2$, the coefficients A and B have the following values: $A = 0.779$, and $B = 0.3713$.

Let us now consider the case of a (μ, σ) normal distribution. The derivation of the formulas is presented in Appendix 1. The ratio of load imbalance costs to the parallel execution time for the first structure of Figure 1 is

$$\psi^{(1)}_{a,K} = \frac{C_X\cdot\mu_X}{K+1}\left(A(a)\cdot S_1(K) - B(a)\cdot S_2(K) - \frac{1}{2A(a)}\cdot S_3(K)\right) \tag{2.29}$$

with $A(a)$, $B(a)$, $S_1(K)$, $S_2(K)$, $S_3(K)$ defined previously.

The results are presented in Figure 3. We see that for relative small values of the coefficient of variation, e.g. for $C_X < 0.05$, the load imbalance increases the execution time only slightly by 10 to 20% even for large computations. For larger coefficients of variation the increase in the execution time grows more rapidly with the number N of processors.

An Upper Bound for a General Distribution. We conclude the discussion of the first structure by deriving an upper bound for $\psi^{(1)}_{a,K}$ for the case when the distribution function of X is continuous, strictly increasing. In this case

$$\psi^{(1)}_{a,K} = \sigma_X \times \frac{D_{(a,K)}}{\sqrt{2}\times(K+1)} \tag{2.30}$$

with $D_{(a,N)}$ given by the following expressions

$$D_{(a,K)} = \begin{cases} D' & \text{if } K = 2K' \quad (K \text{ even}) \\ D' + \left[a^{K'+1} - \frac{1}{a^{K'}}\right] & \text{if } K = 2K'+1 \quad (K \text{ odd}) \end{cases} \tag{2.31}$$

where

$$D' = \frac{1+\sqrt{a}}{a-1}\left(a^{K'} - 1\right)\left[a - \frac{1}{\sqrt{a}a^{K'-1}}\right]. \tag{2.32}$$

According to [8] we have

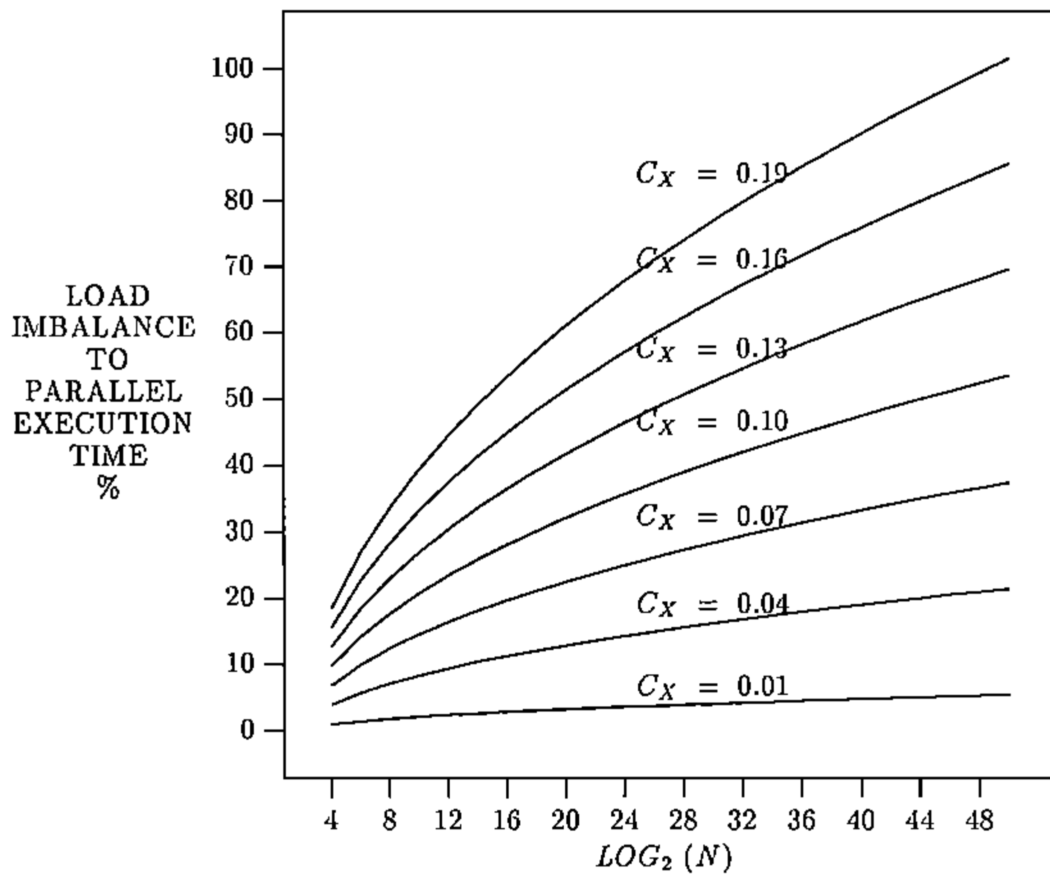


FIGURE 3: The ratio $\psi^{(1)}_{2,k}$ of load imbalance costs to the parallel execution time in absence of load imbalance effects as function of the problem size for the first synchronization structure. The execution time has a (μ, σ) normal distribution with $\mu = 1$.

$$\Delta_i \le C_X \; \frac{I_i - 1}{\sqrt{2I_i - 1}}. \tag{2.33}$$

In our case $I_i = a^i$ and it can be seen easily that

$$\frac{a^i - 1}{\sqrt{2a^i - 1}} \le \frac{a^i - 1}{\sqrt{2a^{i-1}}} \quad \text{for} \quad a > 1. \tag{2.34}$$

But

$$\frac{a^i - 1}{\sqrt{2a^{i-1}}} = \frac{1}{\sqrt{2}} \left[a\sqrt{a^{i-1}} - \frac{1}{\sqrt{a^{i-1}}} \right] \tag{2.35}$$

and

$$\Delta_{a,K}^{(1)} \le \frac{C_X}{\sqrt{2}\; D_{(a,K)}}. \tag{2.36}$$

The Second Synchronization Structure

Consider the second structure presented in Figure 1b. Call $\Delta_{a,N}^{(2)}$ the total load imbalance cost for this structure.

Proposition *For any distribution of the execution time the load imbalance cost for the first synchronization structure is an upper bound for the load imbalance cost of the second structure with the same number of elements.*

$$\Delta_{a,K}^{(2)} \le \Delta_{a,K}^{(1)}. \tag{2.37}$$

The proof is based upon the following observation. For any distribution of the execution time the execution time including any load imbalance effects for the second structure is smaller than the execution time for the first structure. Since the expected execution time of a given processor is the same in both cases it follows that the load imbalance costs for the second case are smaller.

To prove that the execution time for the second structure is always smaller than the one for the first structure let us consider the simple structure presented in Figure 4. Let X_i, X_j, X_k, X_l, X_m and X_n be independent, identically distributed, random variables representing the execution times on processors executing in parallel, subject to synchronization condition as shown in Figure 4.

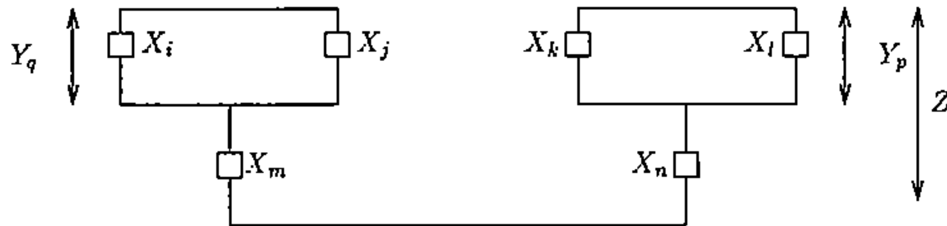


FIGURE 4: An element of the synchronization structure in Figure 1b.

The following expressions give the total execution time, Z, as well as the partial execution times, Y_q, and Y_p :

$$Y_q = \max(X_i, X_j) \tag{2.38}$$

$$Y_p = \max(X_k, X_l) \tag{2.39}$$

$$Z = \max[(X_m + Y_q), (X_n + Y_p)] \tag{2.40}$$

The following inequality follows immediately:

$$\begin{aligned} Z &= \max[(X_m \ + \ \max(X_i, X_j)), (X_n \ + \ \max(X_k, X_l))] \\ &\leq \max(X_m, X_n) \ + \ \max(X_i, X_j, X_k, X_l). \end{aligned} \tag{2.41}$$

But $\max(X_m, X_n) + \max(X_i, X_j, X_k, X_l)$ is precisely the execution time for the first structure with four processors active in the first epoch (the corresponding execution time are X_i, X_j, X_k, X_l), and two active in the second epoch (the corresponding execution time are X_m, X_n).

The results presented in Section 4.1 may be used to estimate the load imbalance costs for the second structure. Note that closed form expressions for the distribution function of the parallel execution time can be derived but it is impractical to construct them.

3 The Effects of Communication Latency Upon Synchronization

3.1 Communication latency on a hypercube

A simplified model of parallel computations with global synchronization was presented in the previous sections. A serious limitation of the model comes from the assumption that in epoch $\mathcal{E}_k$ all I_k PEs, $\pi_1, \ldots, \pi_{I_k}$ start computing precisely at the same time and the epoch terminates when the last processor in the group completes its execution. Communication does not occur instantaneously, on the contrary, the communication latency between any two processors π_i and π_j is substantial.

For example, according to [5] the time to deliver a short message from node i to node j on an NCUBE/1 can be approximated by

$$\delta_{ij} = 261 + 193 d_{ij} \tag{3.1}$$

with

δ_{ij} = the transmission latency in μsec.

d_{ij} = the Hamming distance between node i and node j. For example $d_{14} = 2$ since $1 = 0001$ and $4 = 0100$. The distance d_{ij} represents the number of links to be traversed by a message from the source (i or j) to the destination (j or i) node.

For example, when $d_{ij} = 10$ then $\delta_{ij} \cong 2200$ μsec.

The communication is faster on second generation hypercubes. For example on NCUBE/2 with a 20 MHZ clock, the time to deliver a short message (2 bytes) from node π_i node π_j is approximately [1]

$$\delta_{ij} = 140 + d_{ij} + \frac{12}{20 \cdot 10^6} \simeq 140 + d_{ij} \tag{3.2}$$

with

δ_{ij}	=	the transmission latency in μsec.
140 μsec	=	is the start-up and the close-up time for a connection.
2 μsec	=	is the overhead in routing the packet at every intermediate node along the path.
$20 \cdot$ Mbps	=	the DMA transfer rate.

If $d_{ij} = 10$ then $\delta_{ij} \cong 160\mu$sec. Note that this approximation for δ_{ij} does not take into account possible contention for communication links and/or memory with other nodes, but it is an effective delivery time. It takes into account the software overhead associated with VERTEX, the operating system on NCUBE.

3.2 Synchronization and broadcasting on a hypercube

In this section, we assume that a sub-cube of dimension L is allocated to a computation C which requires global synchronization. Each processor executes the same code, but on different data according to the following pattern. A leader, usually processor π_0, signals the beginning of an epoch and every PE, π_i starts computing and upon termination, signals completion. When π_0 receives completion messages from all π_i, $1 \leq i \leq 2^L - 1$, the next epoch is started.

To analyze quantitatively the effects of communication latency, it is necessary to define precisely the communication patterns involved in global synchronization. In this paper we consider a broadcast-collapse synchronization protocol [18] using the broadcast tree shown in Figure 5. In this protocol, a synchronization epoch starts when π_0 broadcasts a short message signaling the beginning of the epoch. Each processor π_i starts computing as soon as it receives the start-up signal. Each processor sends up a termination signal to its ancestor in the tree when the following two conditions are fulfilled:

(i) It has completed execution.

(ii) It has received a termination signal from all its descendents (if any) in the broadcast tree.

This protocol guarantees that π_0 receives a termination signal if and only if all PEs have signaled termination and there is no contention for communication links, due to signaling of beginning and termination of the epoch.

The tree in Figure 5 has the property that in a cube of order L the number n_ℓ of nodes at level ℓ is

$$n_\ell = \binom{L}{\ell} \quad 0 \leq \ell \leq L. \tag{3.3}$$

In other words, all nodes at distance ℓ from the root are at level ℓ. It follows that if the communication hardware has a fan-out mechanism which allows a node to send a broadcast message to all its immediate descendents at the same time, then the broadcast tree is optimal in the sense that each node receives a broadcast message from the root at the earliest possible time. If such a fan-out is not supported by the hardware, then the following scheme guarantees that the node at the farthest distance from the root receives the broadcast message at the earliest possible time. A node at level ℓ in the broadcast tree (see Figure 5) sends messages to its descendents at level $\ell + 1$, in the order of the depth of the sub-tree routed at that node. For example, π_0 sends the broadcast

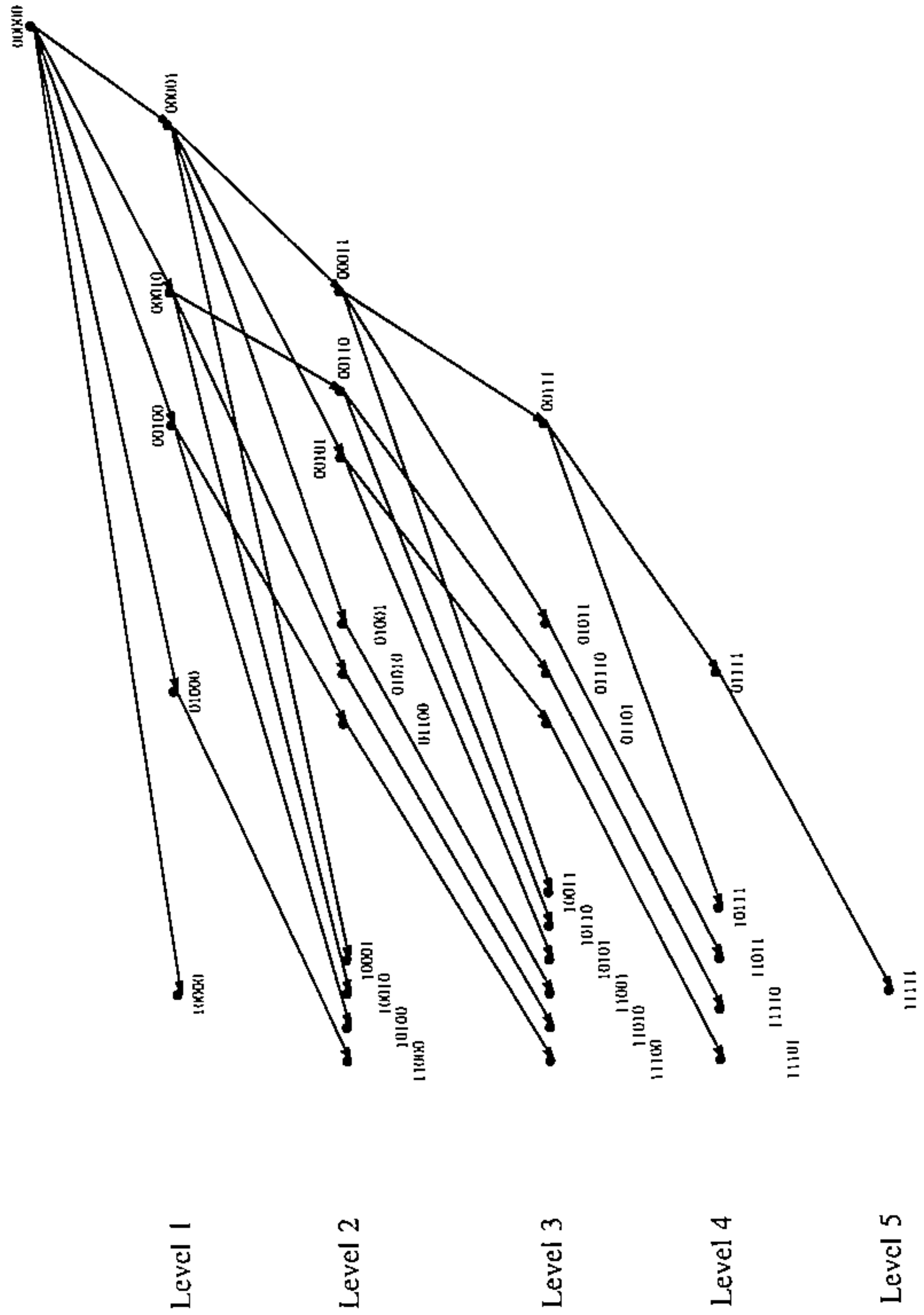


FIGURE 5: A broadcast tree which allows optimal communication time in distributing or collecting messages.

messages in the following order π_1, π_2, π_4, π_8, π_{16}, since the subtrees routed at these nodes have their respective depth 4, 3, 2, 1 and 0.

To implement the synchronization protocol described earlier, each PE at level $\ell > 0$ executes repeatedly the following sequence of steps:

```
read start-up message from ancestor_at_level (ℓ − 1) and pass it to the descendants
    execute computation
      for i = 1, number_descendents_at_level ℓ + 1 do
      begin
        read termination_message (*)
      end
write termination_message to ancestor_at_level ℓ − 1
```

This code assumes a blocking read and a non-blocking write. The generic read statement allows a node to read the termination messages in the order they arrive. If a generic read statement is not available, then the node at level ℓ has to read messages from its descendents at level $\ell + 1$ to minimize blocking. An optimal strategy in this case is to read first from the node with the shortest subtree. For example, π_0 should read in the order π_{16}, π_8, π_4, π_2, π_1, since the corresponding subtrees have the depth 0, 1, 2, 3 and 4 respectively.

Note that in this case the communication time for the collapse mechanism is dependent upon the number of descendents at distance one. A node may have at most L descendents and in our analysis we overestimate the time to read all termination messages as $\delta_1 = L\delta'$ with δ' the time for a read operation which does not block, since the data is already there.

3.3 A first order approximation for the effects of synchronization upon efficiency

To get an estimate of the effect of communication latency in global synchronization upon processor utilization, consider a very simple model based upon the following assumptions.

A1 – The computation C uses a sub-cube of dimension L of a hypercube of dimension N, and there is no interference between the sub-cube allocated to C and other sub-cubes. No messages other than those needed by C are routed through the sub-cube.

A2 – The execution time of the computation allocated to every processor in epoch i is strictly deterministic, $X^i = E$, $0 \le i \le 2^L - 1$.

A3 – A broadcast/collapse mechanism is used for synchronization. To signal the beginning of a synchronization epoch the processor at the root of the broadcast tree, π_0 sends a message of the shortest length at time t^s and the message reaches the n_ℓ processors at level ℓ in the broadcast tree at time $t^\ell = t^s + \ell\delta_0$. To signal the termination of the synchronization epoch, the processors at level L send a message of the shortest length at time time $t^L = t^s + L\delta_0 + E$ and the message is processed by π_0 at time $t^{s'} = t^s + E + L(\delta_0 + \delta_1)$. The timing diagram corresponding to this synchronization protocol is presented in Figure 6 for the case $L = 5$.

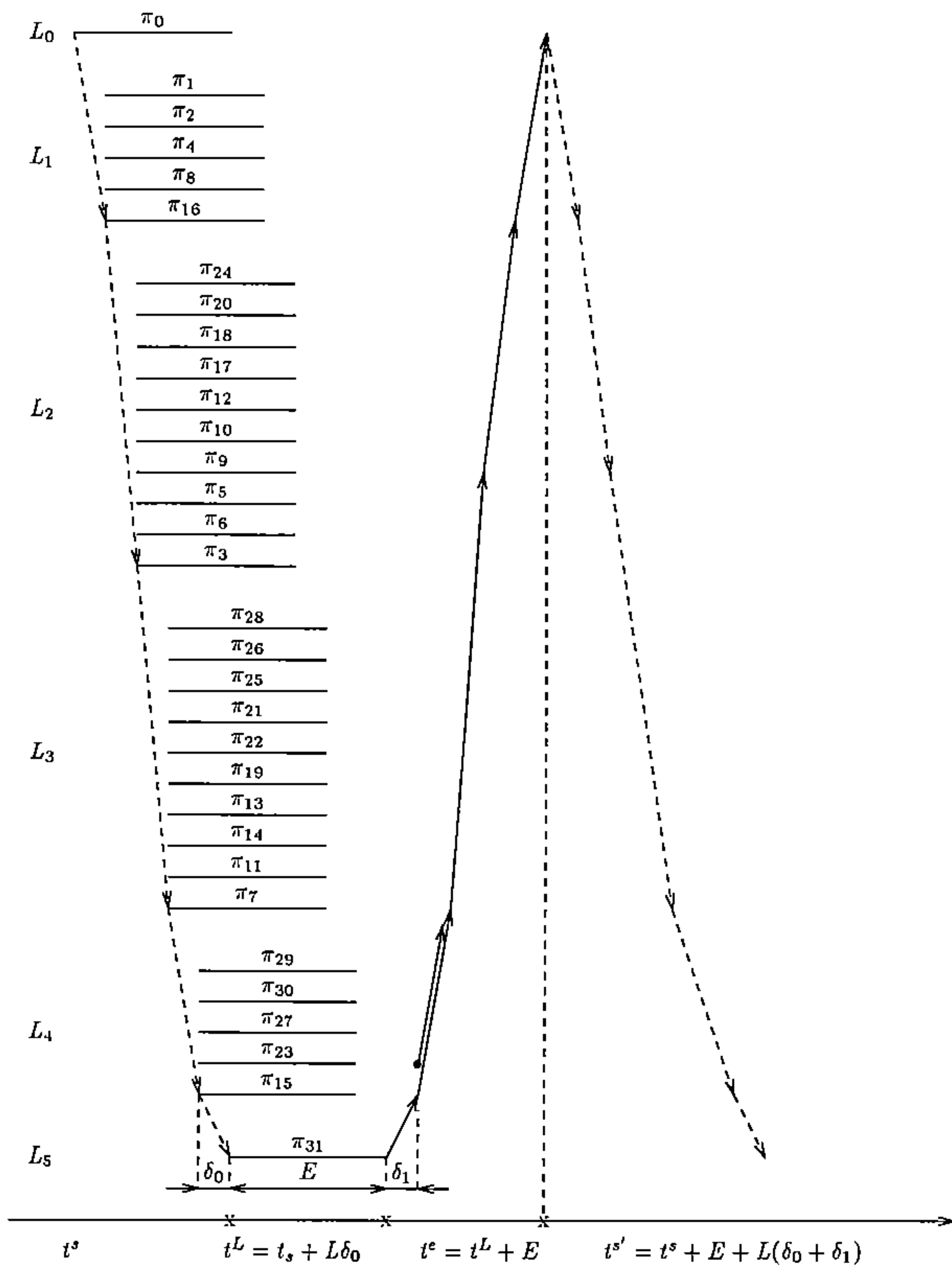


FIGURE 6: A timing diagram for the case $L = 5$. Broadcast time per level is δ_0, collapse time per level is δ_1, and execution time is deterministic E.

Since we have assumed strictly deterministic execution times and no communication interference from other sub-cubes, the synchronization protocol described above guarantees that no blocking due to memory and/or link contention will ever occur, and the *duration of a synchronization epoch* is precisely $T_S = E + L(\delta_0 + \delta_1)$. Call $\delta = \delta_0 + \delta_1$. It follows immediately that when assumptions A1–A3 are true, then the average utilization of any processor is

$$U = \frac{E}{T_S} = \frac{E}{E + L\delta} = 1 - \frac{L\delta}{E + L\delta} \tag{3.4}$$

or

$$U = 1 - \frac{1}{1 + \frac{E}{L\delta}} = 1 - \frac{1}{1+\beta} \tag{3.5}$$

with $\beta = \frac{E}{L\delta}$ a factor describing the computation to communication time ratio.

For example, when $E = 100\delta$, namely when the computation time is two orders of magnitude larger than communication time, then for a cube of order $L = 10$ it follows that $\beta = 10$ and

$$U = 1 - \frac{1}{11} \simeq 0.89.$$

When $E = \delta$ then $\beta = 0.1$ and $U \cong 0.091$.

For the NCUBE/2 we have $\delta \simeq 160$ μsec, while the execution time of the fastest instruction is about 50 nanoseconds. It follows that in order to achieve 90% processor utilization on such a hypercube, each PE has to execute $10^2 \cdot 10^3 = 10^5$ instructions in a synchronization epoch.

The previous analysis can be easily extended to the case when the execution time has a bounded distribution, i.e., when $a \leq X^i \leq b$, and when the broadcast-collapse time per level satisfies $\delta \geq b - a$. In this case it is guaranteed that communication latency hides all the effects of nondeterminancy in execution time. In other words, a processor at level ℓ will always complete its execution no later than the completion messages from all its descendents have arrived, as shown in Figure 6.

Consider processors π_i and π_j with actual execution times a and b, respectively. To complete execution on π_i before receiving all the completion messages, say from π_j, in the worst case scenario we must have the condition $b \leq \delta_0 + \delta_1 + a$ or $\delta \geq b - a$. In this case, the processor utilization is bounded by

$$1 - \frac{1}{1 + \frac{(2^L - 1)a + b}{L}} \leq U \leq 1 - \frac{1}{1 + \beta_b} \tag{3.6}$$

with

$$\beta_b = \frac{b}{L\delta}\,. \tag{3.7}$$

To minimize the inefficiency associated with low values of β_b, the following strategy could be used. Rather than assign equal computations to every PE, assign a higher load to processors at lower levels in the broadcast-collapse tree. For example, Figure 6 suggests that in the case of deterministic execution time, the processors at level ℓ can be kept busy for an additional time equal to

$$\tau_\ell = (L - \ell)\delta. \tag{3.8}$$

The computation time for processors at level ℓ is

$$E_\ell = E + (L - \ell)\delta = E\left[1 + \frac{L - \ell}{L\beta}\right] \tag{3.9}$$

or

$$E_\ell = E\left(1 + \frac{1 - \frac{\ell}{L}}{\beta}\right) \tag{3.10}$$

This simply means that rather than having an equipartition of the data domain, an optimal balanced assignment would mean to assign different computational loads depending upon the position of the processor in the broadcast-collapse tree.

With this approach all n_ℓ PEs at level ℓ have a utilization

$$U_\ell = 1 - \frac{\ell\delta}{E + L\delta} = 1 - \frac{\ell}{L}\frac{1}{1+\beta}. \tag{3.11}$$

The average utilization is

$$U^{opt} = \frac{1}{2^L}\sum_{\ell=0}^{L} n_\ell U_\ell \tag{3.12}$$

and substituting for U_ℓ we have

$$\begin{aligned} U^{opt} &= \tfrac{1}{2^L}\textstyle\sum_{\ell=0}^{L} n_\ell U_\ell = 1 - \tfrac{1}{2^L}\sum_{\ell=0}^{L}\binom{L}{\ell}\left[1 - \tfrac{\ell}{L}\tfrac{2}{1+\beta}\right] \\ &= 1 - \frac{\sum_{\ell=0}^{L}\ell\binom{L}{\ell}}{L2^L}\frac{1}{1+\beta} \end{aligned} \tag{3.13}$$

or

$$U^{opt} = 1 - \frac{L2^{L-1}}{L2^L}\frac{1}{1+\beta} = 1 - \frac{1}{2}\frac{1}{1+\beta} \tag{3.14}$$

To evaluate the advantage of this approach, consider the case when

$$E = \delta.$$

In this case, using an equipartition of the load, the average processor utilization is

$$U = 1 - \frac{L}{L+1}. \tag{3.15}$$

with $\beta = E/L\delta = 1/L$. Using the optimal load partition the utilization is

$$U^{opt} = 1 - \frac{1}{2}\frac{L}{1+\beta} = 1 - \frac{1}{2}\frac{L}{L+1}. \tag{3.16}$$

For $L = 10$, $U = 0.091$ and $U^{opt} = 0.5455$. The corresponding speed ups are

$$S = 2^L \cdot U \cong 92$$

$$S^{opt} = 2^L \cdot U^{opt} \cong 546.$$

For the case $L = 10$, $E = 100\delta$ considered earlier, we have

$$U = 0.89,\ S = 911$$

$$U^{opt} = 0.976,\ S^{opt} = 1000$$

3.4 Expected processor utilization in iterative methods

The model discussed so far does not take into account communication delays due to the need to update boundary values. If one considers a 2-D problem, we assume that then at the beginning of each iteration every PE has to exchange boundary values with at most 4 PEs holding neighboring data subdomains. This effect can be captured by adding a communication time δ_c to the computation time E. Then the average processor utilization is

$$U = \frac{E}{L\delta + \delta_c + E} \tag{3.17}$$

with δ_c and E previously defined and

$$\delta_c = q \times \tau \times \alpha \tag{3.18}$$

with

- q – The number of neighboring data subdomains,
- τ – The time to exchange one boundary value with a neighboring subdomain at distance $d = 1$,
- α – A factor ≥ 1 determined by the mapping strategy and describing the effects of the distance between nodes upon the communication delay.

Then U becomes

$$U = 1 - \frac{L\delta + q\tau\alpha}{L\delta + q\tau\alpha + E} = 1 - \frac{1}{1+\beta} \tag{3.19}$$

with β defined as

$$\beta = \frac{E}{L\delta + q\tau\alpha} \tag{3.20}$$

3.5 A scheme with self-synchronization

Consider now a computation in which global synchronization should occur at every iteration. We present a scheme in which global synchronization occurs only every R-th iteration and during the intermediate iterations each PE attempts a *self synchronization.* This scheme is illustrated in Figure 7. At time t_0 the leader (π_0) sends a start-up signal. All PEs defer starting the computation until time $t_0' = t_0 + L\delta_0$.

The scheme assumes that each PE knows the expected duration E of its computation time per iteration, and has a good bound Δ on its average load imbalance amount. Thus the PEs are expected to complete their execution by time $t_0'' = t_0' + E + \Delta$ and at this time all PEs start exchanging boundary values. This communication period has a duration of $\delta_c = q \times \tau \times \alpha$ with $\alpha \geq 1$, the factor described above, and τ the time required to exchange one boundary value. For each PE there are q other PEs containing adjacent data subdomain. For 2D problems we have $q = 4$ and for 3D problems, $q = 8$. At time t_1 this communication period terminates and a new iteration begins. This scheme could be improved slightly, for example, by having a PE which has not completed the computation by time t_0'' but has finished its communication before time t_1 start the next iteration earlier.

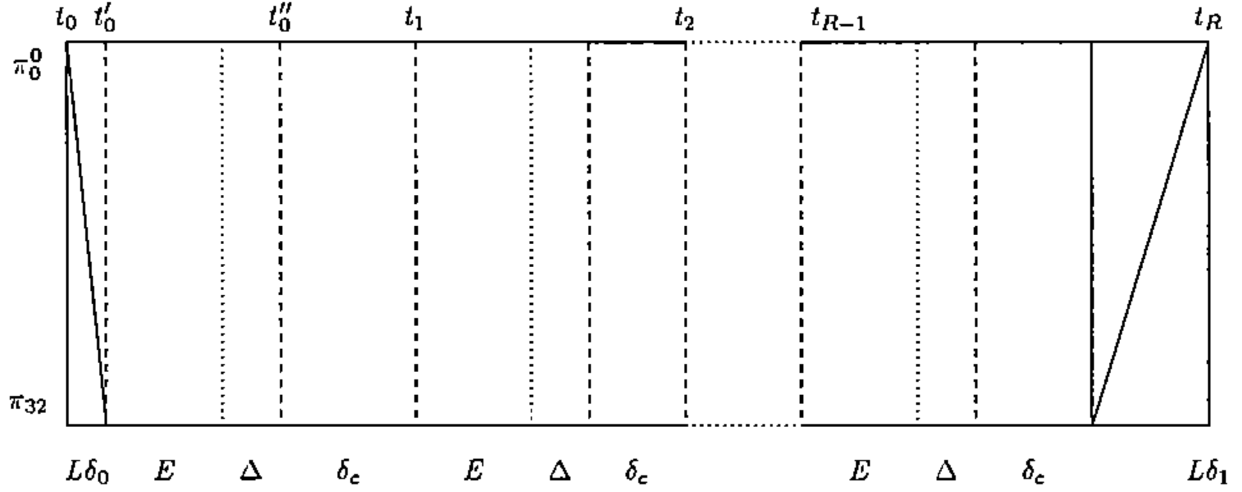


FIGURE 7: The timing diagram for the self-synchronization scheme. True synchronization is done only every R-th iteration. Here E = expected compute time, Δ = bound on load imbalance, δ_c = time to exchange boundary values.

The processor utilization for this scheme is approximated by the following expression

$$U = \frac{R\mu}{L\delta + R(E + \Delta) + Rq\alpha\tau} \tag{3.21}$$

or

$$U = 1 - \frac{L\delta + R(\Delta + q\alpha\tau)}{L\delta + R(E + \Delta + q\alpha\tau)}. \tag{3.22}$$

Often $q\alpha\tau >> \Delta$, and the utilization becomes in this case

$$U = 1 - \frac{1}{1+\beta} \tag{3.23}$$

with

$$\beta \cong \frac{L\delta + R(E + q\alpha\tau)}{L\delta + Rq\alpha\tau} \tag{3.24}$$

Table 2 shows values of speed up predicted by this model of the self-synchronization scheme. We fix the values $L = 10$, $q = 4$, $\alpha = 2$ and $\tau = \delta$ and vary R, E and Δ. As before, we use the ratio E/δ and also set $\Delta = \gamma E$. The formula for speed up is then

$$\begin{aligned} S &= \frac{2^L R(E/\delta)}{L\delta + R(q\alpha\tau/\delta + (E/\delta)(1+\gamma))} \\ &= \frac{1024R(E/\delta)}{10 + R(8 + (E/\delta)(1+\gamma))} \end{aligned} \tag{3.25}$$

Some extreme values for the speed up are:

1. As $E \to \infty$ then $S \to 1024/(1+\gamma)$.
2. As $R \to \infty$ then $S \to 1024(E/\delta)/(8+(E/\delta)(1+\gamma))$.
3. As $R \to \infty$ with $(E/\delta) = 1$, then $S \to 1024/(9+\gamma) \approx 102$ to 114.
4. With $(E/\delta) = 1$, $R = 1$, then $S = 1024/(19+\gamma) \approx 50$ to 54.

Note that $R = 1$ corresponds to ordinary synchronization as discussed in the previous section.

TABLE 2: Speedups for a self-synchronized iteration on a 1024 processor NCUBE with $L = 10$, $q = 4$, $\tau = \delta$. The values R = number of self synchronized iterations, (E/δ) = computation to communication ratio, and (Δ/E) = variation of computation are varied as indicated.

Imbalance $(\Delta/E) = 0$, E/δ Ratio

R	1	5	20	100	inf
1	54	223	539	868	1023
4	89	331	672	927	1023
10	103	366	707	940	1023
25	109	382	722	945	1024
inf	114	394	732	949	1024

Imbalance$(\Delta/E) = 0.1$, E/δ Ratio

R	1	5	20	100	inf
1	54	218	512	800	930
20	89	320	631	850	930
100	102	354	661	861	931
500	108	369	674	865	931
inf	113	380	683	868	931

Imbalance $(\Delta/E) = 0.4$, E/δ Ratio

R	1	5	20	100	inf
1	53	205	446	649	731
4	86	293	532	681	731
10	99	320	554	688	731
25	105	333	563	690	731
inf	109	342	569	692	731

Imbalance$(\Delta/E) = 1.0$, E/δ Ratio

R	1	5	20	100	inf
1	52	183	354	470	512
20	82	250	406	487	512
100	93	270	418	490	512
500	99	279	424	492	512
inf	103	285	427	493	512

The most significant observation from Table 2 is that there is a regime of operation where self-synchronization is quite effective for increasing performance. This occurs when the E/δ is relatively small, say between 1 and 20. Thus for $E/\delta = 5$ and $\Delta/E = 0.1$, we see that self-synchronization can increase the speed up from 218 to 380, or about a 75% improvement. As E/δ becomes large, the synchronization cost is small anyway, so self-synchronization only helps a little. When E/δ is very small (1 or less), the best speed up is already very low so that while the improvement due to self-synchronization is perhaps a factor of 2, the resulting efficiency is still very low.

Note that two of the factors kept constant in Table 2 also affect the speed up substantially. Thus, if $\alpha = 1$ (the optimum value – and often achievable) or $\alpha = 1.5$, then we have the following effects for $E/\delta = 5$ and $\Delta/E = 0.1$. For $\alpha = 1.5$, the speed up goes from 244 to 466 as R goes from 1 to infinity; for $\alpha = 1.0$, the speed up goes from 263 to 539 (an improvement of over 100%). This is plausible because reducing α means making the computation more local, which improves the efficiency. The second factor is L, if $L = 13$ instead of 10, then again the computation becomes more local and the efficiency plus the advantage of self-synchronization improves. For the same case $(E/\delta) = 5$ and $\Delta/E = 0.1$, the speed up increases from 1546 to 3034 or about a 96% improvement due to self-synchronization. With $L = 13$ and $\alpha = 1$, the speed up for this case increases from 1821 to 4311 or about a 137% improvement.

4 Experimental Results

An experiment to study the performance of iterative methods on a distributed memory system is described in detail in [8]. The experiment uses the parallel ELLPACK (PELLPACK) system developed at Purdue [7], running on a 128 processor NCUBE/1. The TRIPLEX tool set [6], is used to monitor the execution and to collect trace data.

The experiment monitors the execution of the code, implementing a Jacobi 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 PELLPACK environment, attempts to assign to every PE an equal amount of computation. The experiment was conducted by taking a problem of a fixed size and repeating the execution with a number of PEs ranging from 2 to 128.

The experiment monitors communication events and permits the determination of the time spent by every computation assigned to a PE, called in the following a *thread of control*, in any state. A thread of control can be either *active* (or computing), or *performing an I/O operation*, (either reading or writing), or in a *blocked state*, (waiting while attempting to read data from another PE). Communication is done strictly by broadcasting and the broadcast-collapse mechanism uses two balanced binary trees rooted at π_0 and π_1, respectively. Every five iterations a convergence test is done to ensure that the computation converges. The time is measured in ticks, and 1 tick = 0.167 milliseconds.

In the following, we discuss the case of a rectangular domain and a 50×50 grid with 64 processors used to solve the problem [8]. Figures 8, 9 and 10 present histograms of the computing times for processors on each of the levels of the broadcast tree. The groupings are PEs 0 and 1 in Figure 8a-8b, PEs 2 and 3 in Figures 8c-8d, PEs 4 to 7 in 9a-9b, PEs 8 to 15 in 9c-9d, PEs 16 to 31 in 10a-10b, and PEs 32 to 63 in 10c-10d. Figures 11, 12 and 13 present histograms of the blocking time with the same grouping of processors.

The computing time for PE 0 is displayed in Figures 8a and 8b. The first shows that most of the computing time is done in slices of 50 ticks or less, but a few slices (less than 2%) take considerably longer. We suspect that these relatively long computing intervals correspond to the

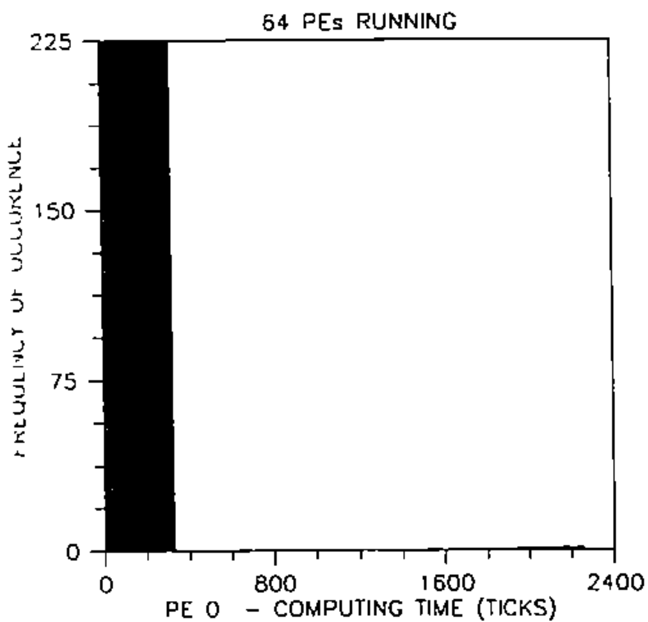
convergence tests done every five iterations. A close-up of the computing intervals with length in the 0 to 50 ticks range is presented in Figure 8b, which shows that about 80% of the computing intervals last less than 10 ticks. The same approach is taken to other groups, the first diagram in each pair presents a histogram of all the computing intervals observed for the group and the second diagrams zooms upon the region with computing intervals of 50 ticks or less.

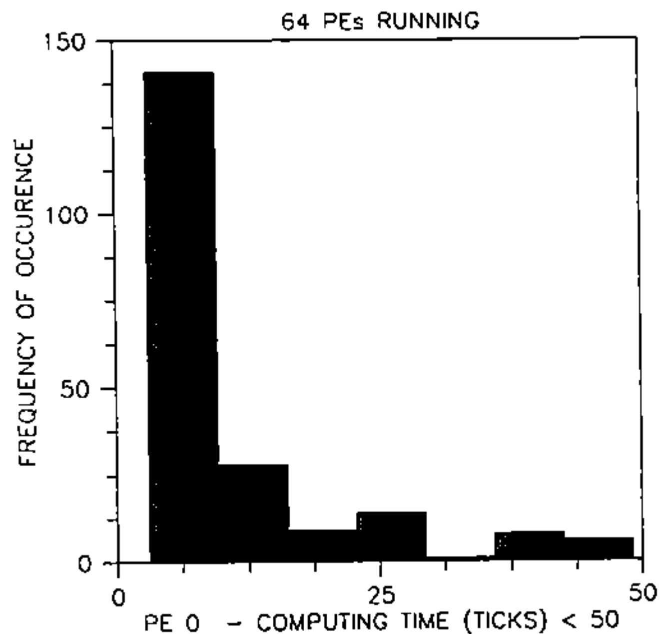
The following trends are common for all groups of processors. About 98% of the computing intervals are of 50 ticks or less. About 2% of all intervals are higher than 50 ticks, but the duration of these few relatively long computing periods decreases as the level of the group in the broadcast tree increases. For example, these long periods are of about 2000 ticks for level 0 and close to 400 for level 5. Probably close to 80% of all computing intervals are the length of 10 ticks or less. The expected length of a computing interval decreases as the group is located at a higher level in the broadcast tree. The distribution of the computing time is bounded, but it is not well approximated with either a uniform or a normal distribution.

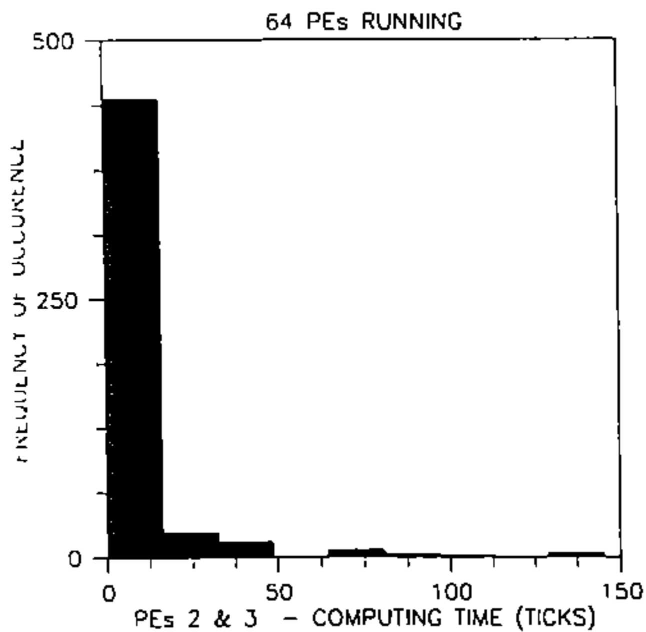
Let us now discuss the blocking time intervals. A thread of control enters a blocked state as a result of a READ operation when the data requested are not available in the local buffer associated with the link on which the message is expected. Since communication time, in particular the blocking time, depends upon the actual communication hardware, we have defined and measured the *algorithmic blocking.* The algorithmic blocking is a measure of the amount of time the demand for data at the consumer processor precedes the actual generation of data by the producer processor. The algorithmic blocking is measured as the interval from the instance when a READ is issued by a consumer PE, until the corresponding WRITE is issued by the producer PE. If the WRITE precedes the READ, then the algorithmic blocking is considered to be zero.

The blocking time is always larger than the algorithmic blocking. The non-algorithmic blocking, defined as the difference between the blocking time and the algorithmic blocking time, is a measure of the communication latency. Congestion of the communication network leads to large non-algorithmic blocking times. Figures 11, 12 and 13 present pairs of histograms for the blocking and algorithmic blocking for several groups of processors. Figures 11a and 11b show the blocking time for PE 0 and for PEs 2 and 3. PE 0 exhibits blocking for relatively short periods of time of 100 ticks or less. The effects of communication latency are visible, the algorithmic blocking is about 80% of all blocking intervals of less than 50 ticks, while blocking times larger than 50 ticks occur in about 50% of all cases. PEs 2 and 3 experience longer blocking periods as shown in Figure 11b. This trend continues for PEs 4 to 7, Figures 11c and 11d; and PEs 8 to 15, Figures 12a and 12b; and PEs 16 to 31, Figures 12c and 12d. For these cases, the histograms of all blocking times in the group and the histogram of blocking times less than 200 ticks are shown in pairs.

Figure 13 presents in more detail the 32 to 63 PE group. The overall histogram in Figure 13a is as before and the histogram of the non-algorithmic blocking time is also given in Figure 13d. Their counterparts are given in Figures 13c and 13b for the case when the blocking time is less than 200 ticks. Again, we observe an anomaly, namely in a few instances a fairly large blocking interval occurs. A plausible explanation is that these effects are due to the start-up and termination.







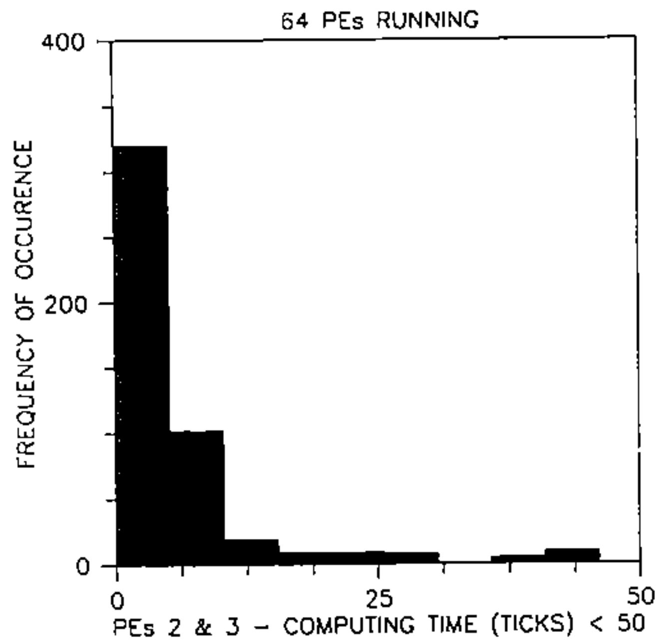
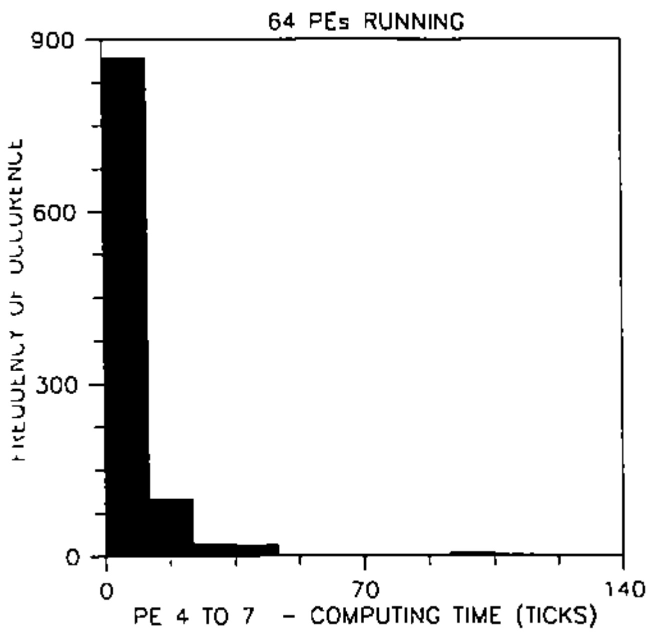
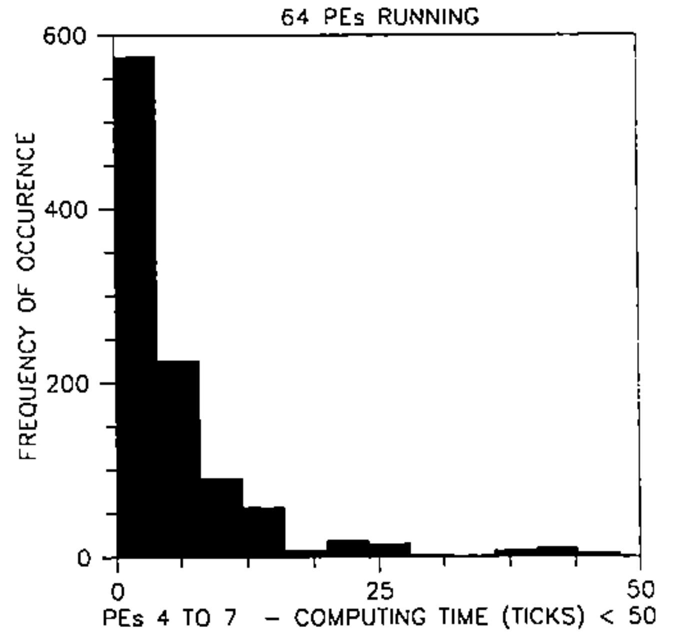
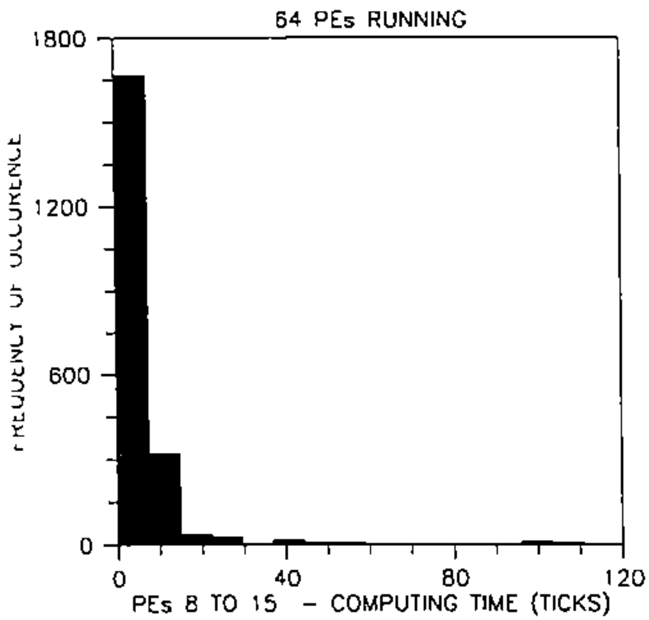


Figure 8. Histograms of computing time for PE 0, (a) and (b), and PEs 2 and 3, (c) and (d).







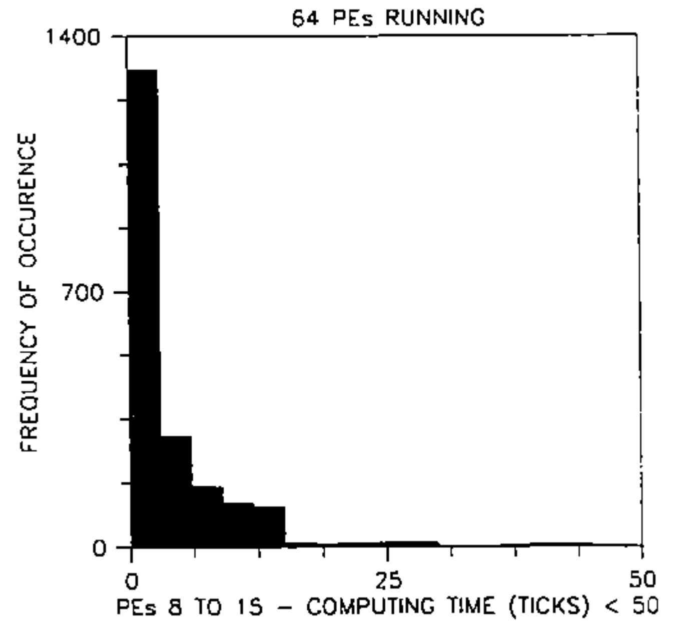
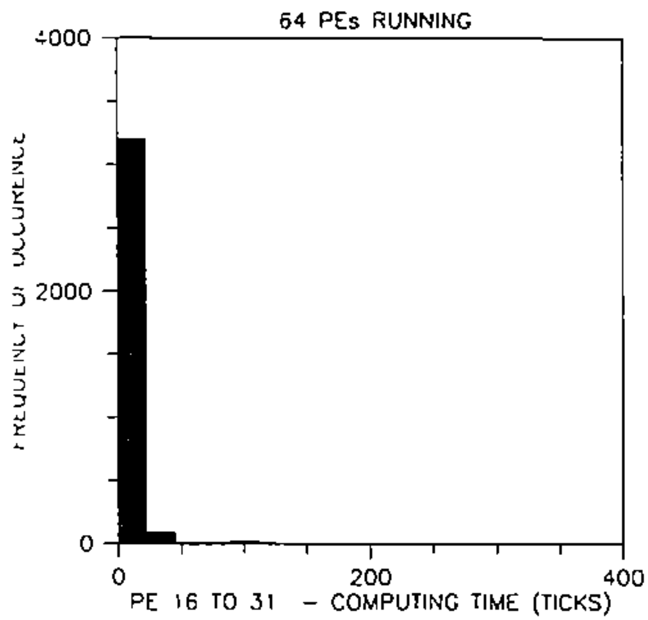
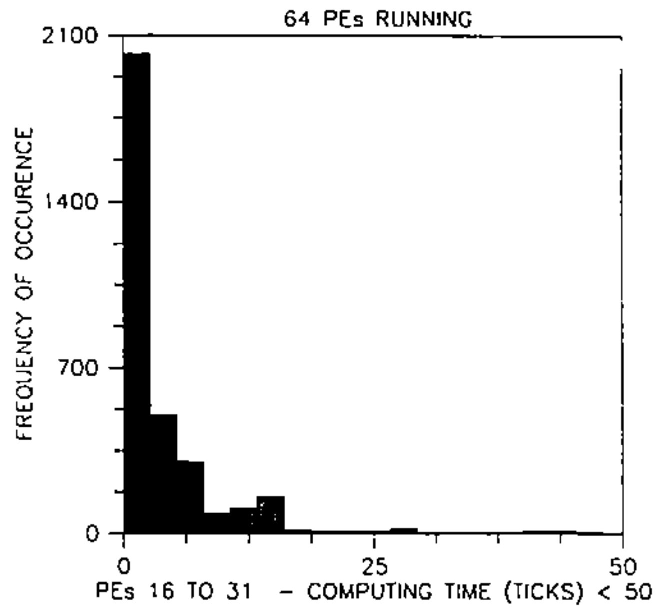
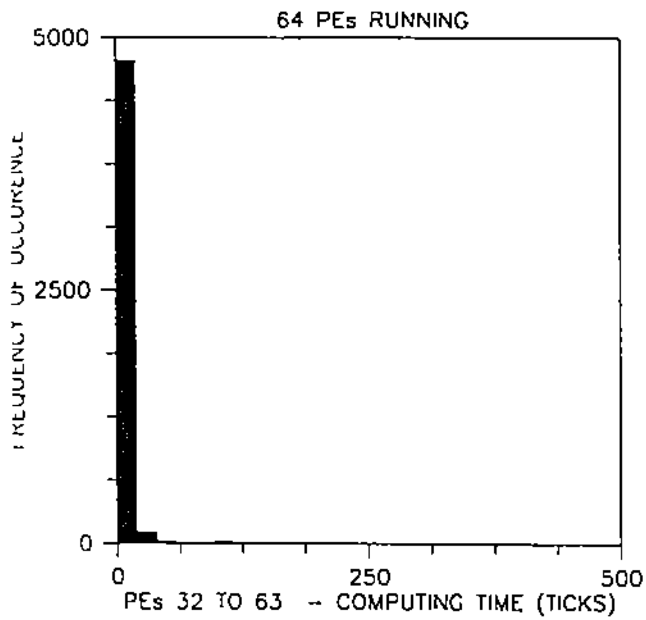


Figure 9. Histograms of computing time for PEs 4-7, (a) and (b), and PEs 8-15, (c) and (d).







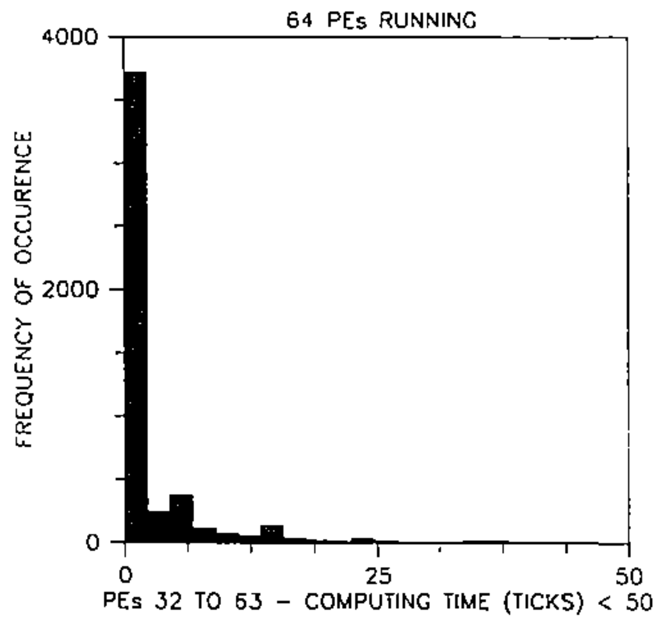
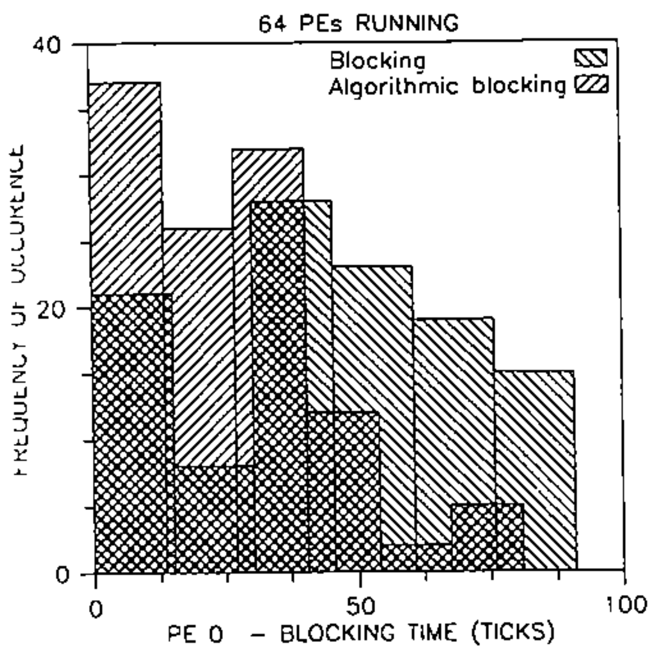
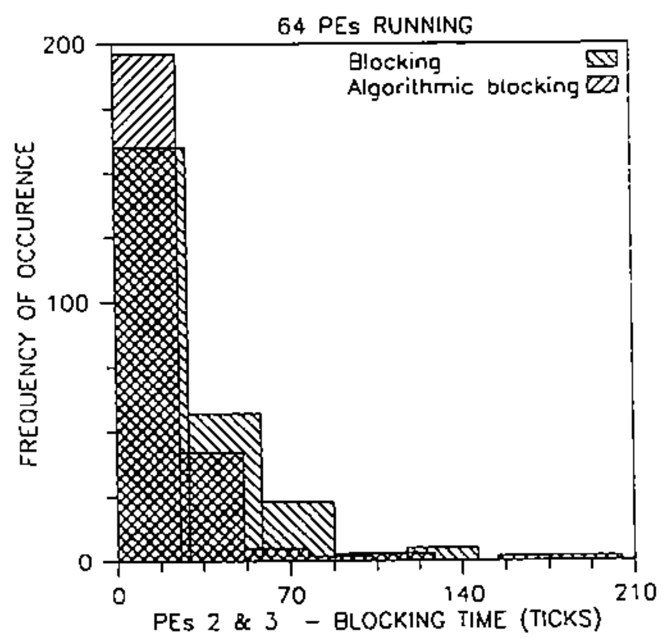
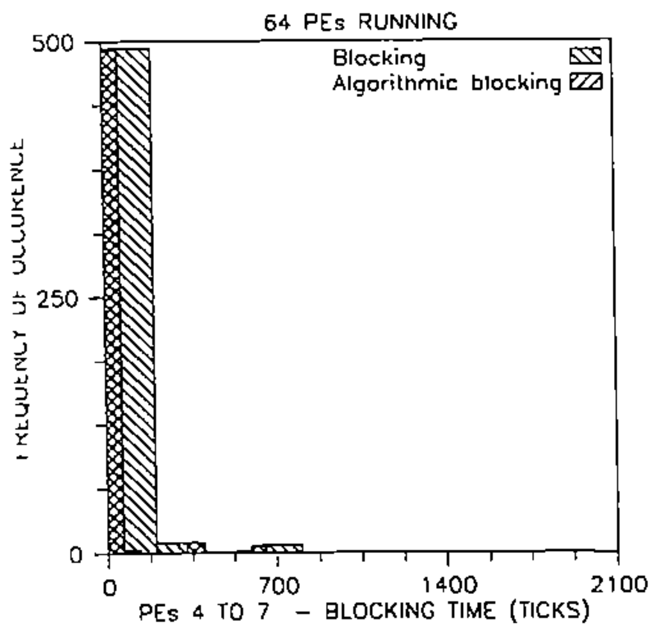


Figure 10. Histograms of computing time for *PE*s 16-31, (a) and (b), and *PE*s 32-63, (c) and (d).







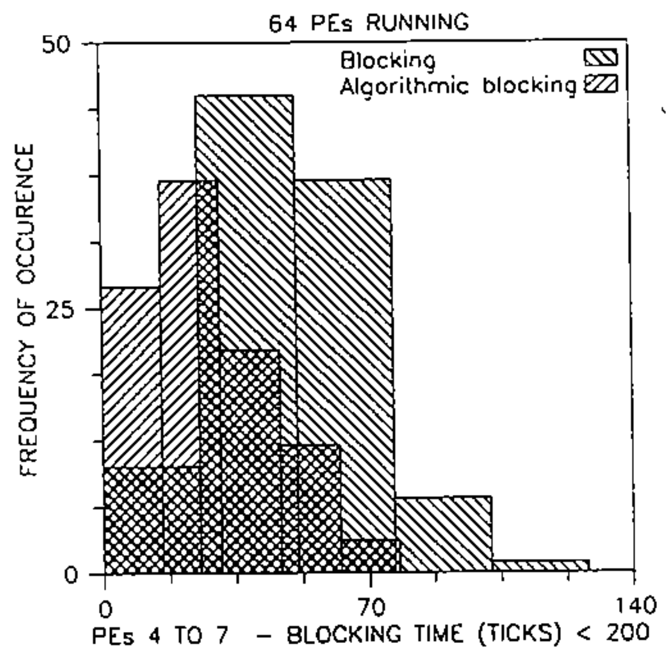
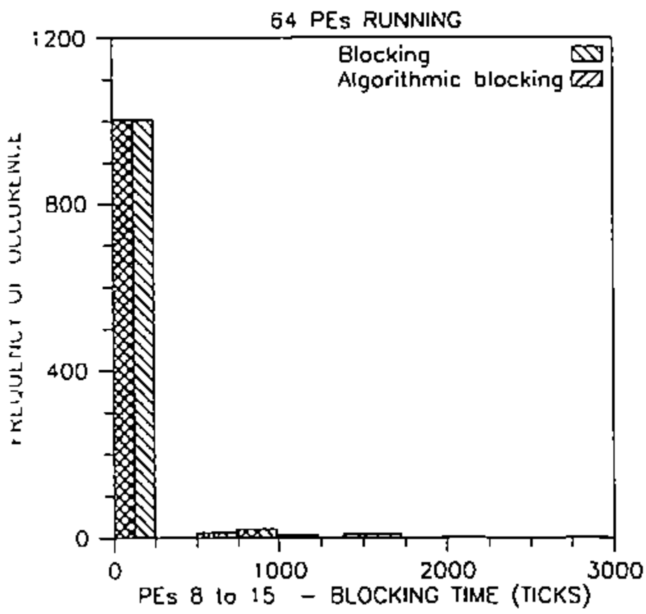
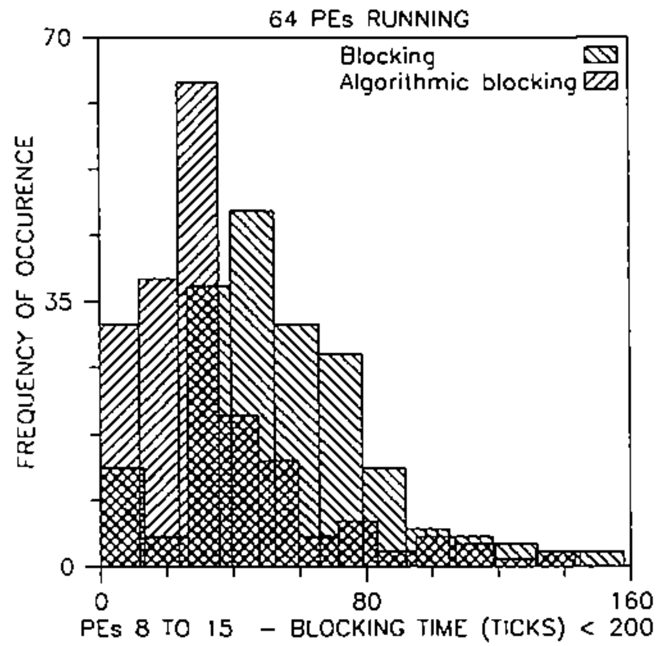
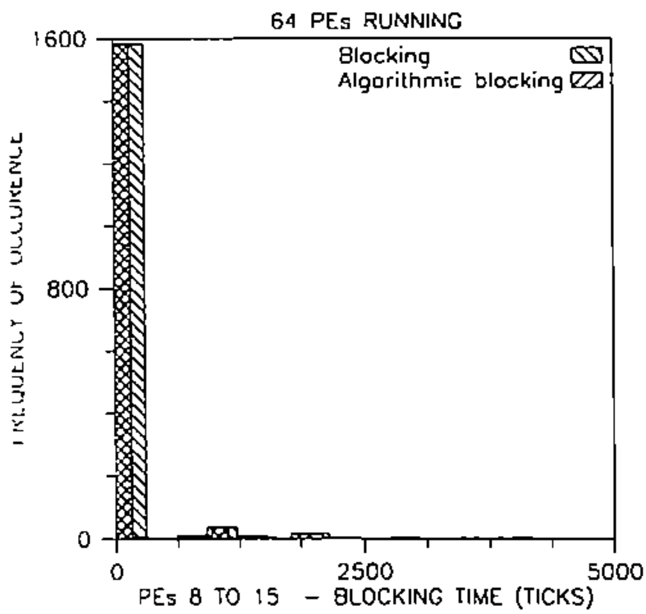


Figure 11. Histograms showing both total blocking and algorithmic blocking times for PE 0, (a); PEs 2-3, (b); and PEs 4-7, (c) and (d).







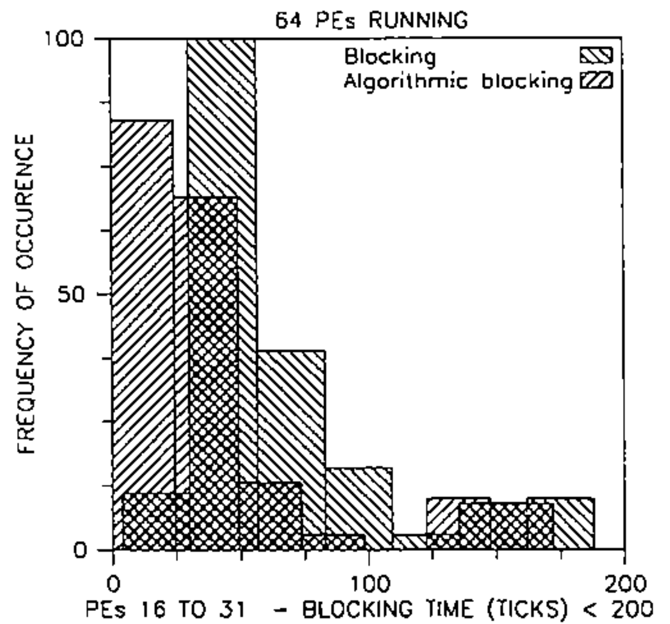
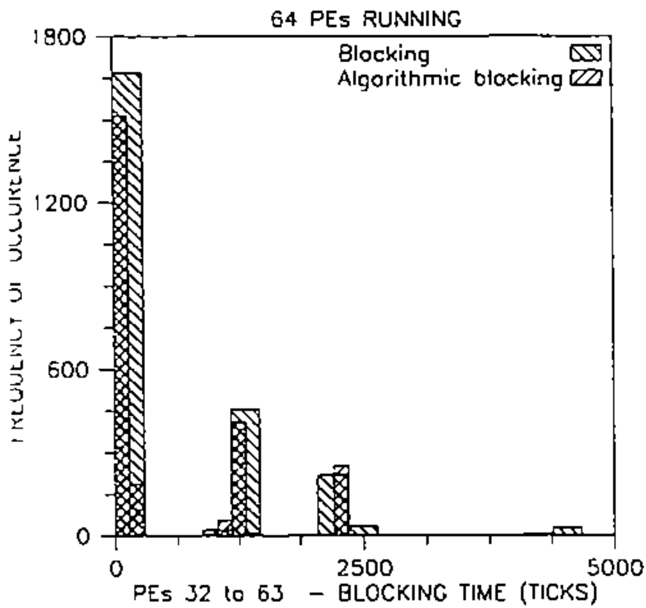
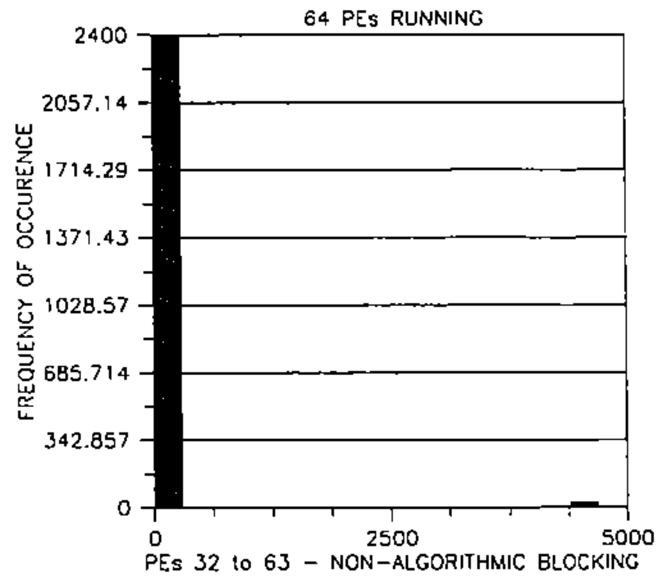
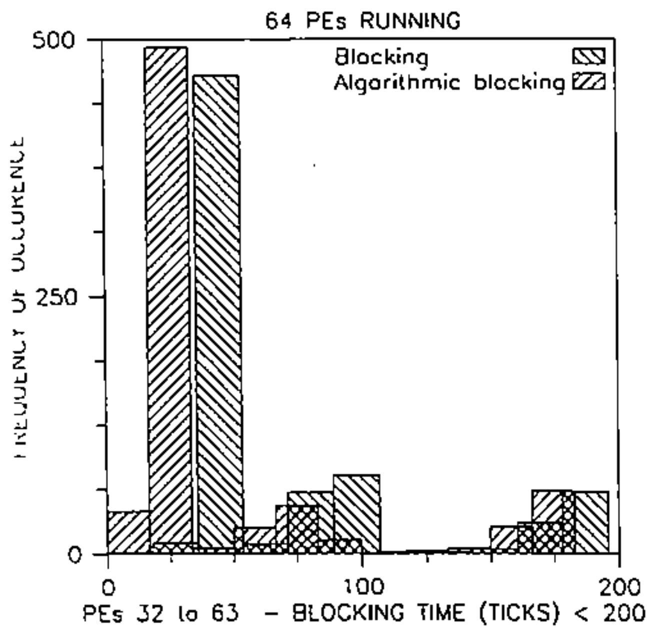


Figure 12. Histograms showing both total blocking and algorithmic blocking for PEs 8-15, (a) and (b), and for PEs 16-31, (c) and (d).







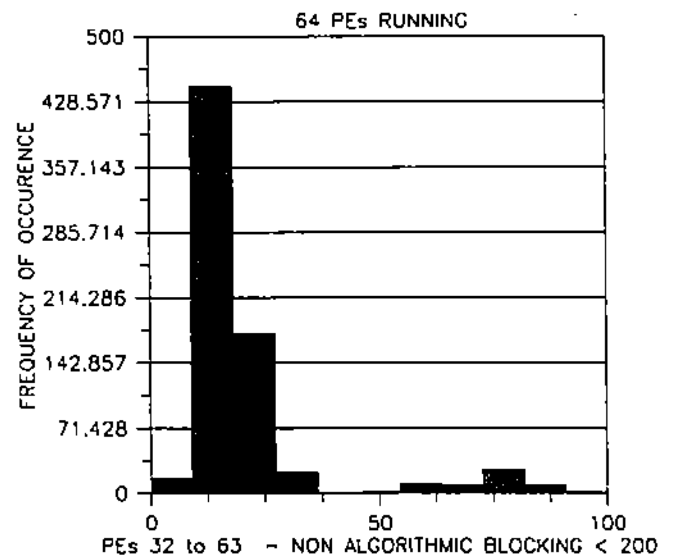


Figure 13. Histograms showing both total blocking and algorithmic blocking for PEs 32-63, (a) and (c), and non-algorithmic blocking for PEs 32-63, (b) and (d).

5 Conclusions

The paper reports results on the performance of iterative methods on distributed memory MIMD systems. High communication costs and dynamic load imbalance are responsible for the low speed up observed in practice for computations which require global synchronization.

Architectural and/or algorithmic approaches may be considered to improve the performance. Architectural solutions, likely to be found in future generations of distributed memory MIMD machines, are to provide either a fast, dedicated communication sub-system for handling short messages for synchronization, or some form of shared memory for semaphores.

An algorithmic solution to the problem is to reduce the number of synchronization points in iterative methods. Yet, the numerical convergence is guaranteed only in case of synchronized execution. Therefore, it is desired to eliminate explicit synchronization but have different threads of control coordinate their execution implicitly. The self-synchronization method we propose achieves this by estimating the time when all threads of control have completed an iteration. This approach requires an understanding of the effects of the dynamic load imbalance due to non-deterministic execution time, to estimate this time.

This paper is devoted to the study of algorithmic solutions. Section 2 of the paper presents an analysis of the dynamic load imbalance. A load imbalance factor is defined and its upper bounds for different distributions of the execution time are computed. Section 3 analyzes the effects of the communication latency and produces a closed form expression for the expected processor utilization which takes into account the communication latency. Then an algorithm for self-synchronized execution is given. Section 4 presents an experiment involving the parallel ELLPACK system in which detailed information concerning the behavior of all threads of control was collected. These data do not conform to model developed, there is a bi-model behavior. The first mode involves small delays consistent with our model and are due to expected variations in balance and data dependencies. The second mode involves very long delays due to the global synchronizations required. The long delays (and there are also some intermediate ones) prevent us from validating our model in any statistical sense, but the data does suggest that our model is appropriate for the effects of local variations and irregularities in SPMD computations.

Acknowledgements

This work was supported in part by the Strategic Defense Initiative through ARO grants DAAG07-86-K-0106 and DAAL03-90-0107, by the National Science Foundation by grant CCR-8619817 and by the NATO grant 891007. The experimental work reported in Section 4 from [8], is the result of a collaboration with E.A. Vavalis who contributed significantly to performing the experiment and analyzing the results.

APPENDIX 1 - The load imbalance factor for the first structure in case of a normal distribution.

First consider a standard normal distribution. In this case $\mu = 0$ and $\sigma = 1$ which leads to

$$\Delta_i \approx (2\log I_i)^{1/2} - \frac{1}{2}(2\log I_i)^{-1/2}(\log 4\pi - 2C) - \frac{1}{2}(2\log I_i)^{-1/2}(\log\log I_i) \qquad (A.1)$$

Since $I_i = a^{K-i}$, Δ_i becomes:

$$\Delta_i(a,K) \approx A(a)(K-i)^{+1/2} - B(a)(K-i)^{-1/2} \; - \; \frac{1}{2A(a)} \, (K-i)^{-1/2} \log(K-i)$$

$$A(a) = (2\log a)^{1/2} \qquad (A.2)$$

$$B(a) \;=\; \frac{1}{2A(a)} \, [\log 4\pi - 2C + \log\log a]. \qquad (A.3)$$

Then

$$\Delta^{(1)}_{a,N}(a,K) = \sum_{i=1}^{K-1} \Delta_i \approx A(a) \cdot S_1(K) - B(a) \cdot S_2(K) \; - \; \frac{1}{2A(a)} \, S_3(K) \qquad (A.4)$$

with

$$S_1(K) = \sum_{i=1}^{K-1} (K-i)^{1/2} = \sum_{i=1}^{K-1} (i)^{1/2} \qquad (A.5)$$

$$S_2(K) = \sum_{i=1}^{K-1} (K-i)^{-1/2} = \sum_{i=1}^{K-1} (i)^{-1/2} \qquad (A.6)$$

$$S_3(K) = \sum_{i=1}^{K-1} (K-i)^{-1/2} \log(K-i) = \sum_{i=1}^{K-1} i^{-1/2} \log i \qquad (A.7)$$

According to Ramanujan [11]:

$$S_1(K) = C_1 \; + \; \frac{2}{3}\,(K-1)\sqrt{K-1} \; + \; \frac{1}{2}\,\sqrt{K-1} +$$
$$\frac{1}{6} \; \left[\left\{ \sqrt{K-1} + \sqrt{K} \right\}^{-3} + \left\{ \sqrt{K} + \sqrt{K+1} \right\}^{-3} + \ldots \right] \qquad (A.8)$$

with

$$C_1 = -\,\frac{1}{4\pi} \; \left(\frac{1}{1\sqrt{1}} + \frac{1}{2\sqrt{2}} + \frac{1}{3\sqrt{3}} + \ldots \right) \qquad (A.9)$$

The asymptotic expansion for large value of K can be shown to be

$$S_(K) = C_1 \; + \; \frac{2}{3}\,(K-1)\sqrt{K-1} \; + \; \frac{1}{2}\,\sqrt{K-1} +$$
$$\frac{1}{\sqrt{K-1}} \; \left(\frac{1}{24} \; - \; \frac{1}{1920(K-1)^2} \; + \; \frac{1}{9216(K-1)^4} \; \ldots \right) \qquad (A.10)$$

Then $S_2(K)$ is

$$S_2(K) = C_0 + \sqrt{K-1} \; + \; \frac{1}{2\sqrt{K-1}} -$$
$$-\,\frac{1}{2} \; \left\{ \; \frac{\{\sqrt{K-1} + \sqrt{K}\}^{-3}}{\sqrt{K(K-1)}} + \frac{\{\sqrt{K} + \sqrt{K+1}\}^{-3}}{\sqrt{K(K+1)}} + \ldots \right\} \qquad (A.11)$$

with

$$C_0 = -(1+\sqrt{2})\left(\frac{1}{\sqrt{1}} - \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} - \frac{1}{\sqrt{4}} + \ldots\right) \quad (A.12)$$

To evaluate $S_3(K)$ note that

$$\sum_{i=1}^{\infty} i^{-x} \log i = -\zeta'(x) \quad (A.13)$$

for $1 \le x \le \infty$. $\zeta(x)$ is Riemann's zeta function. It follows that

$$S_3(K) \le -\zeta'(\sqrt{2}) \approx 307.8223572 \quad (A.14)$$

When $N = 2^K$, the coefficients A and B have the following values: $A = 0.779$, and $B = 0.3713$. Finally

$$\psi_{2,K}^{(1)} = 0.779\frac{S_1(K)}{K+1} - 0.3713\frac{S_2(K)}{K+1} - 0.6444\frac{S_2(K)}{K+1} \quad (A.15)$$

Consider the general case of a (μ, σ) normal distribution. The load imbalance costs for a synchronization epoch with I_i processors active is

$$\Delta_i \cong \sigma_X[(2\log I_i)^{1/2}) - \frac{1}{2}(2\log I_i)^{-1/2}(\log\log I_i + \log 4\pi - 2C)] \quad (A.16)$$

Consequently the ratio of load imbalance costs to the parallel execution time for the first structure is

$$\psi_{2,K}^{(1)} = \frac{\mu_X \sum_{i=1}^{K-1} \Delta_i}{\mu_X(K+1)} = \frac{C_X \cdot \mu_X}{K+1}\left(A \cdot S_1(K) - B \cdot S_2(K) - \frac{1}{2A} \cdot S_3(K)\right) \quad (A.17)$$

with S_1, S_2, S_3 previously defined.

Literature

[1] Berry, R, "Private communication",

[2] Fox, G., et al., "Solving problems on concurrent processors", Prentice Hall, (1988).

[3] Gustafson, J.L., Montry, G.R. and Brenner, R.E., "Development of parallel methods for a 1024-processor hypercube", *SIAM J. Sci. Stat. Comp.*, Vol. 9, (1988) pp. 609-638.

[4] Glowinski, R., *Domain Decomposition Methods for Partial Differential Equations*, SIAM Publications, Philadephia, (1991).

[5] Heller, D.E., "Performance measurements on the NCUBE/10 multiprocessor", Technical Report, C S Department, Shell Development Company, Houston, (1988).

[6] Houstis, E.N., and Rice, J.R., "Parallel ELLPACK: An expert system for parallel processing of partial differential equations", *Math. Comp. Simulation*, Vol. 31, (1989) pp. 497-507.

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

[8] Marinescu, D.C., Rice, J.R., and Vavalis, E.A., "Performance of iterative methods for distributed memory multiprocessors", *Proc IMACS 13-th World Congress*, (1991), in press.

[9] Marinescu, D.C., and Rice, J.R., "Synchronization of nonhomogeneous parallel computations", *Parallel Processing for Scientific Computing*, (G. Rodrigue, ed.), SIAM, (1989), pp. 362-367.

[10] Marinescu, D.C., and Rice, J.R., "The effects of communication latency upon synchronization and load balance on a hypercube", *Proc. 5-th Intl. Parallel Processing Symp.*, IEEE Press, (1991), pp. 18–25.

[11] Marinescu, D.C., and Rice, J.R., "Multilevel asynchronous iterations for PDE's", in *Iterative Methods for Large Linear Systems*, (D.R. Kincaid and Linda J. Hayes, eds.), Academic Press, (1990), pp. 193-214.

[12] Ramanujan, S., *Collected papers*, Chelsea Publishing Company, 1962.

[13] Rice, J.R., and Marinescu, D.C., "Analysis of a two level asynchronous algorithm for PDEs", in *Aspects of Computations on Asynchronous Parallel Processors*, (M. Wright, ed.), North Holland, (1989), pp. 23-33.

[14] Rice, J.R., and Marinescu, D.C., "Analysis and modeling of Schwartz splitting algorithms for elliptic PDEs", in *Advances in Computer Methods for Partial Differential Equations*, VI (Stepleman and Vishnevetsky, eds.), IMACS, Rutgers University, (1987), pp. 1-6.

[15] Rice, J.R., "Parallel methods for partial differential equations", in *The Characteristics of Parallel Computation*, (Jamieson, et. al., eds.), MIT Press, (1987), pp. 209-231.

[16] Rodrique, G., *Parallel Processing for Scientific Computing*, SIAM Publications, Philadelphia, (1989).

[17] Rossmann, et al., "Molecular replacement and real space averaging", Purdue University (in preparation).

[18] Saad, Y. and Schultz, M.H., "Data communication in hypercubes", Report YALEU/DCS/RR-428, Yale University, (1985).

*** "The NCUBE 6400 Processor Manual", NCUBE, Beaverton, (1988).