1990

The Effects of Communication Latency Upon Synchronization and Dynamic Load Balance on a Hypercube

Dan C. Marinescu

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

Report Number:
90-1032

THE EFFECTS OF COMMUNICATION LATENCY UPON SYNCHRONIZATION AND DYNAMIC LOAD BALANCE ON A HYPERCUBE

Dan C. Marinescu
John R. Rice

CSD-TR-1032
September 1990

The Effects of Communication Latency Upon Synchronization and Dynamic Load Balance on a Hypercube*

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

Abstract

In this paper we discuss the effects of the communication latency upon the dynamic load balance for computations which require global synchronization are discussed. Experimental results from the study of the performance of iteration methods on an NCUBE 1 are presented.

1 Introduction

In this paper, we are concerned with the effects of synchronization upon the performance of iterative methods on hypercubes. We restrict our discussion to the case when all PEs execute the same program, but on different data; the so-called SPMD paradigm. The SPMD paradigm is widely used today and it will probably continue to be so for large applications using tens of thousands of PEs, since it is fairly difficult to write and debug many different programs for the processing elements of a large system. The SPMD execution implies that the same program and different data sub-domains are loaded in the local memory of the distributed memory multiprocessor.

To obtain a large speedup, it is necessary to maintain a high processor utilization. The amount of time processors are idle needs to be kept as low as possible, 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 static load balancing does not provide a guarantee of *dynamic load balancing*. Due to data dependencies, the actual flow of control of different PEs will be different, different PEs will execute different sequences of instructions. Even the execution time of an instruction is data dependent. 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.

Synchronization is an important aspect of computing using message passing in a distributed memory multiprocessor system. There are numerous cases where synchronization is necessary. For example, iterative methods require synchronization since a processor has to exchange boundary values with its neighbors, but only after *all of them* have finished their computation on the current iteration. Communication on a distributed memory machine is fairly expensive, the time to

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

exchange a short message between two nearest neighbors is typically equivalent to the time to execute 10^2 floating point operations. It follows that synchronization, an operation which requires a consensus of all processors and involves the exchange of a large number of messages, is a source of considerable inefficiency. Observe that if the execution time on all PEs were perfectly equal, then synchronization would be unnecessary. It would suffice to start all the computations at the same time and all iterations will stay synchronized. Data dependencies make this approach unfeasible.

If the study of data dependencies allows us to compute the load imbalance factor Δ, (defined in [11]), then methods to reduce the effects of synchronization can be considered. For example, a method of self synchronization discussed in Section 6 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 a synchronization takes place only every Rth iteration.

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

2 Communication Latency on a Hypercube

A simplified model of parallel computations with global synchronization is presented in [6]. A serious limitation of the model comes from the assumption that in epoch $\mathcal{E}_k$ all the I_k processing elements, $\pi_1, \ldots, \pi_{I_k}$ 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, so concurrent start ups are not possible. On the contrary, the communication latency between any two processors π_i and π_j is substantial.

For example, according to [2], the time to deliver a short message from node i to node j on an NCUBE/1 can be approximated by

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

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.

When $d_{ij} = 10$ then we have $\delta_{ij} \cong 2200$ 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]

$$\delta_{ij} = 140 + 2 d_{ij} + 2 \times \frac{12}{20 \cdot 10^6}$$

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 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 using the broadcast tree shown in Figure 1. 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 1 has the property that in a cube of order L the number n_ℓ of nodes at level ℓ is

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

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 1) 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

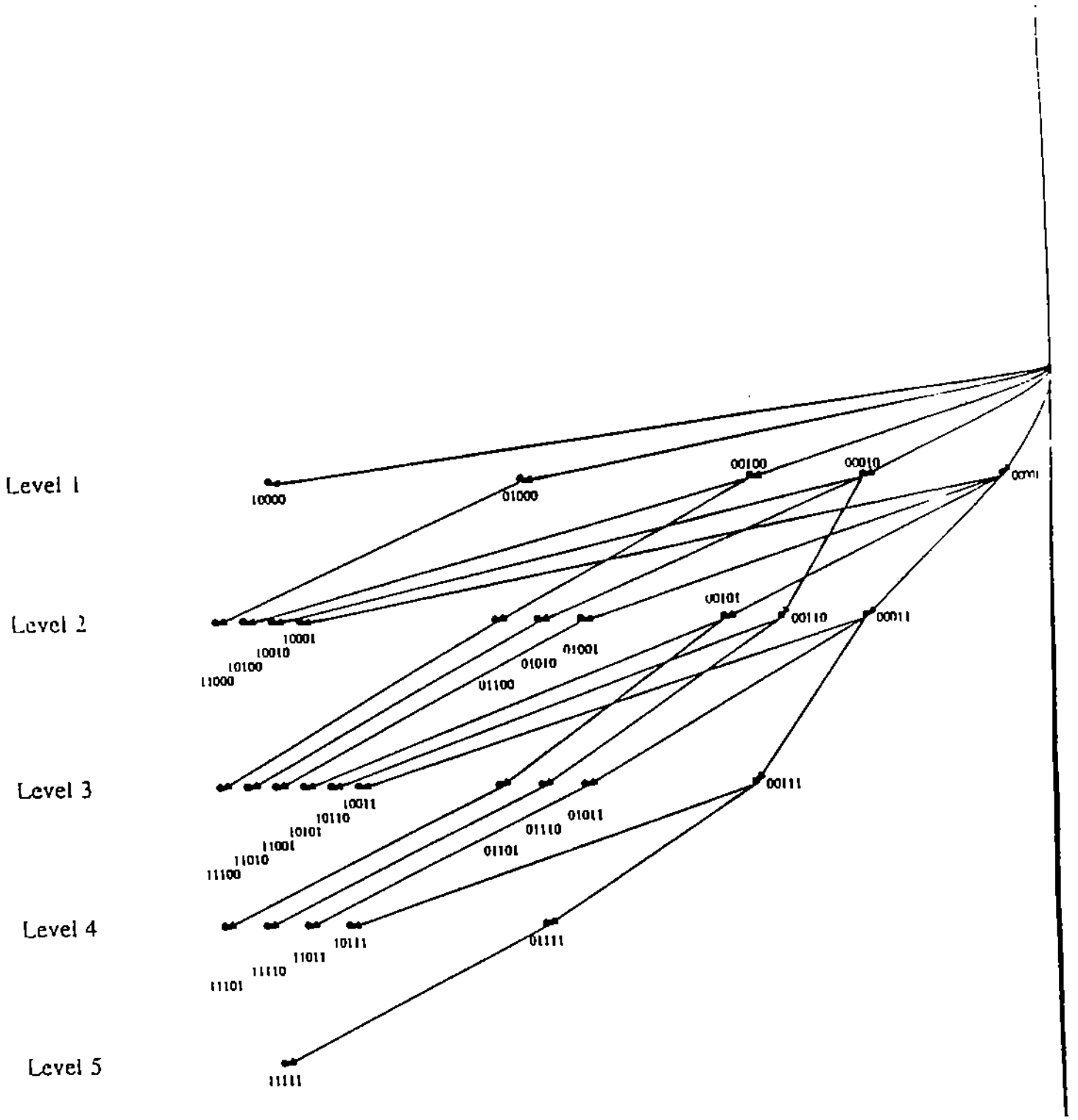


FIGURE 1: A broadcast tree which allows optimal communication time in distributing or collecting messages.

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)
    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.

4 A First Order Approximation for the Effects of Synchronization Upon Efficiency

To estimate 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 a constant E, $0 \leq i \leq 2^L - 1$.

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 2 for the case $L = 5$.

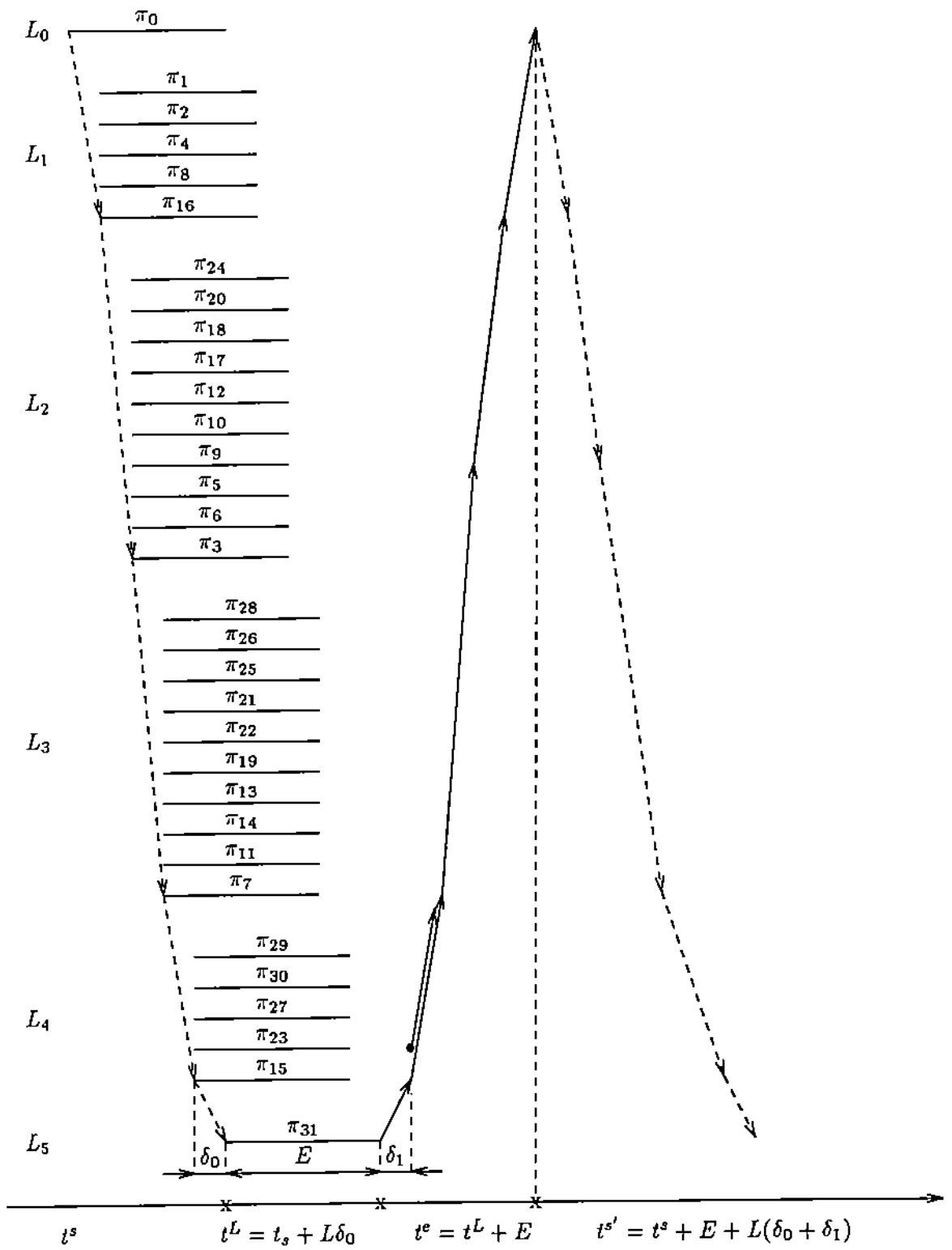


FIGURE 2: A timing diagram for the case $L = 5$. Broadcast time per level is δ_0, collapse time per level is δ_1, and the execution time is E.

Since we have assumed constant 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

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

or

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

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 about $10^2 \cdot 10^3 = 10^5$ instructions in a synchronization epoch.

The previous analysis can be easily extended to the case when the execution time has a bounded stastical distribution, i.e., when the execution time X^i on π_i satisfies $a \leq X^i \leq b$, 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 2.

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

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

with

$$\beta_b = \frac{b}{L\delta}.$$

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 2 suggests that in the case of constant execution time E the processors at level ℓ can be kept busy for an additional time equal to

$$\tau_\ell = (L - \ell)\delta.$$

The computation time for processors at level ℓ could then be

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

or

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

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

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

The average utilization is

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

and substituting for U_ℓ we have

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

or

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

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

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

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

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

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

$$U = 0.89,\ S = 911$$
$$U^{opt} = 0.976,\ S^{opt} = 1000$$

5 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

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

with δ_c and E previously defined and

$$\delta_c = q \times \tau \times \alpha$$

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

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

with β defined as

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

6 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 3. 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

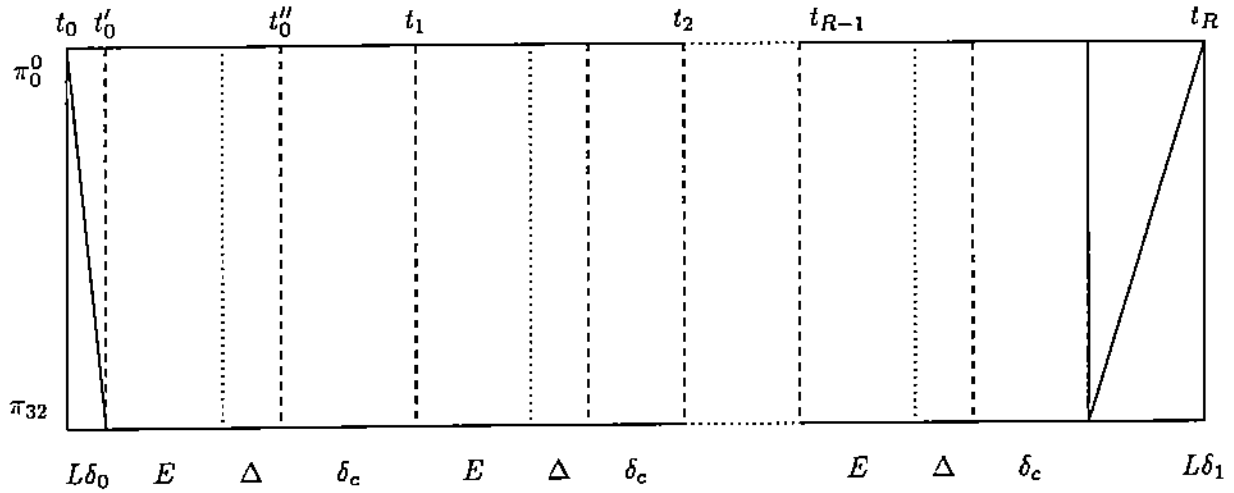


FIGURE 3: The timing diagram for the self-synchronization scheme. True synchronization is done only every R-th iteration. Here E = expected compute time, Δ = bound on load imbalance, δ_c = time to exchange boundary values.

not completed the computation by time t_0'' but has finished its communication before time t_1 start the next iteration earlier.

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

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

or

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

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

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

with

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

Table 1 shows values of speedup 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 speedup is then

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

Some extreme values for the speedup 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.

The most significant observation from Table 1 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 speedup 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 speedup 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 1 also affect the speedup 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 speedup goes from 244 to 466 as R goes from 1 to infinity; for $\alpha = 1.0$, the speedup goes from 263 to 539 (an improvement of over 100%).

TABLE 1: 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

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 speedup increases from 1546 to 3034 or about a 96% improvement due to self-synchronization. With $L = 13$ and $\alpha = 1$, the speedup for this case increases from 1821 to 4311 or about a 137% improvement.

7 Experimental Results

An experiment to study the performance of iterative methods on a distributed memory system is described in detail in [5]. The experiment uses the parallel ELLPACK (PELLPACK) system developed at Purdue [3], running on a 128 processor NCUBE/1. The TRIPLEX tool set [4], 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 [5].

First, we 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 4, 5 and 6 present pairs of histograms for the blocking and algorithmic blocking for several groups of processors. Figures 4a and 4b 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 4d.

This trend continues for PEs 4 to 7, Figures 4c and 4d; and PEs 8 to 15, Figures 5a and 5b; and PEs 16 to 31, Figures 5c and 5d. 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 6 presents in more detail the 32 to 63 PE group. The overall histogram in Figure 6 is as before and the histogram of the non-algorithmic blocking time is also given in Figure 6d. Their counterparts are given in Figures 6c and 6d 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.

Literature

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

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

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

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

[5] Marinescu, D.C., J.R. Rice and E. Vavalis, "Performance of iterative methods for distributed memory processors", CSD-TR 979, Computer Sciences Department, Purdue University, 1990.

[6] Marinescu, D.C. and J.R. Rice, "Synchronization and load imbalance effects in distributed memory multiprocessor systems", CSD-TR 1000, Computer Sciences Department, Purdue University, July 1990.

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

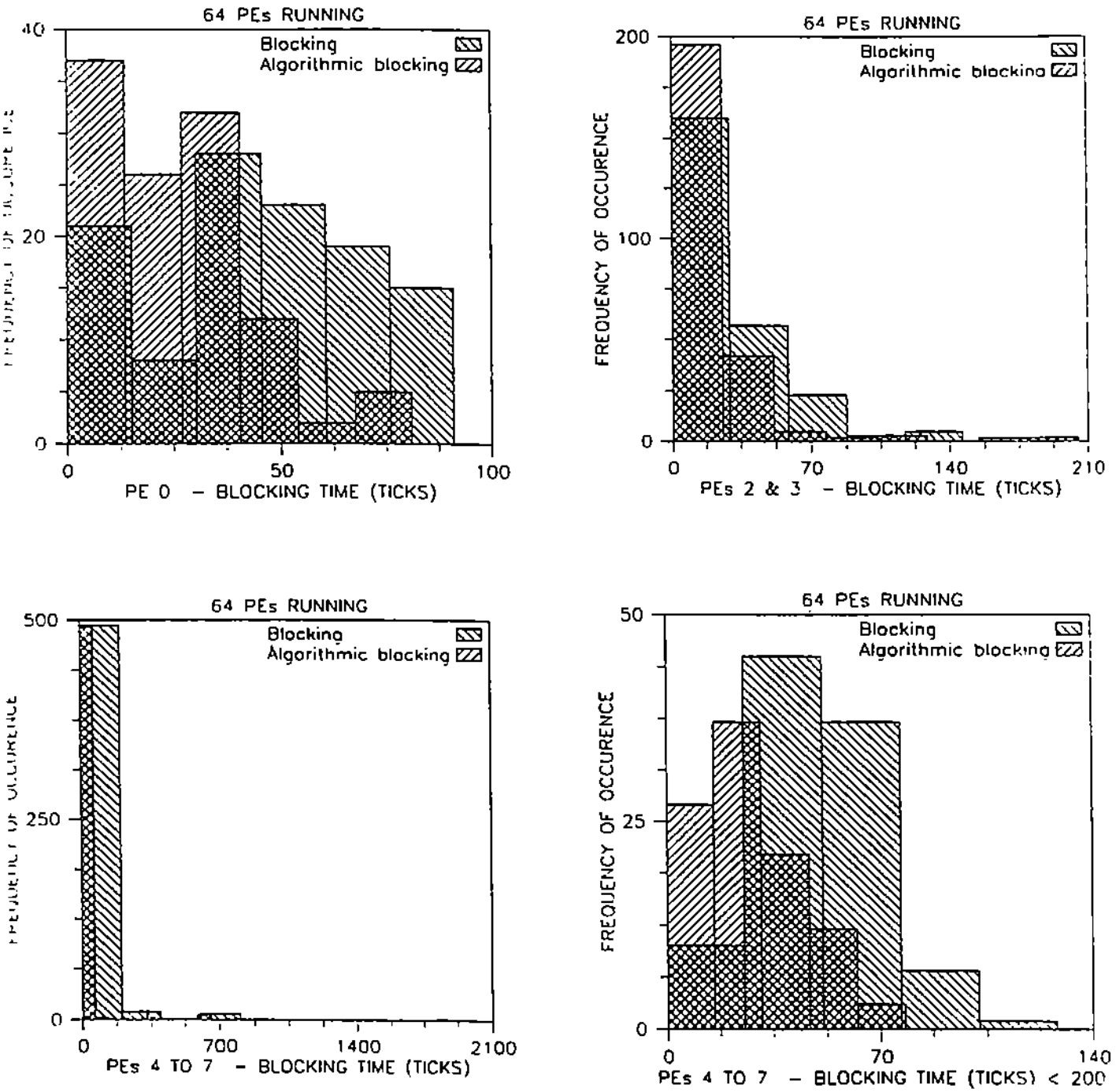


Figure 4. Histograms showing both total blocking and algorithmic blocking times for PE 0, (a); PEs 2-3, (b); and PEs 4-7, (c) and (d).

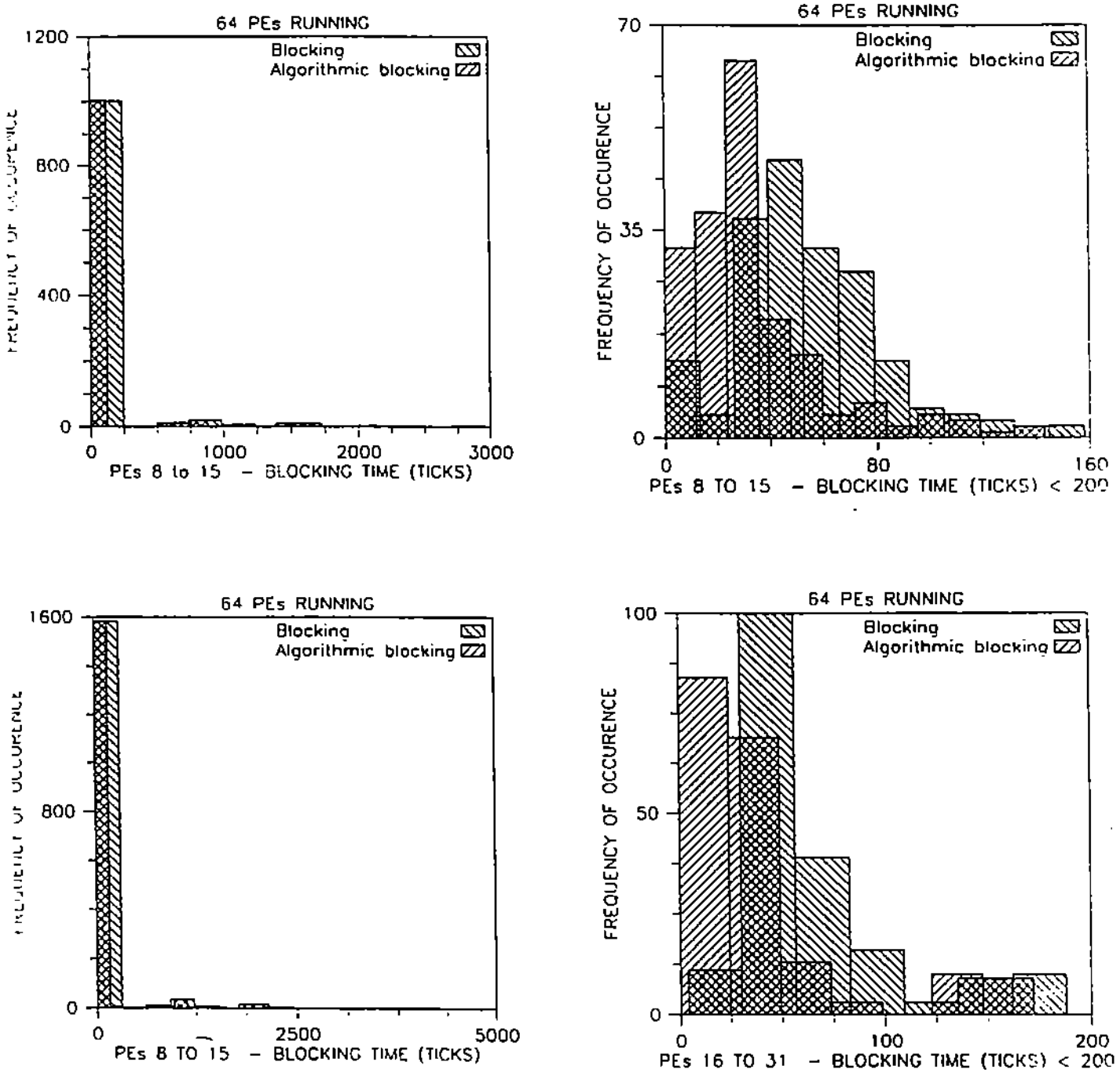
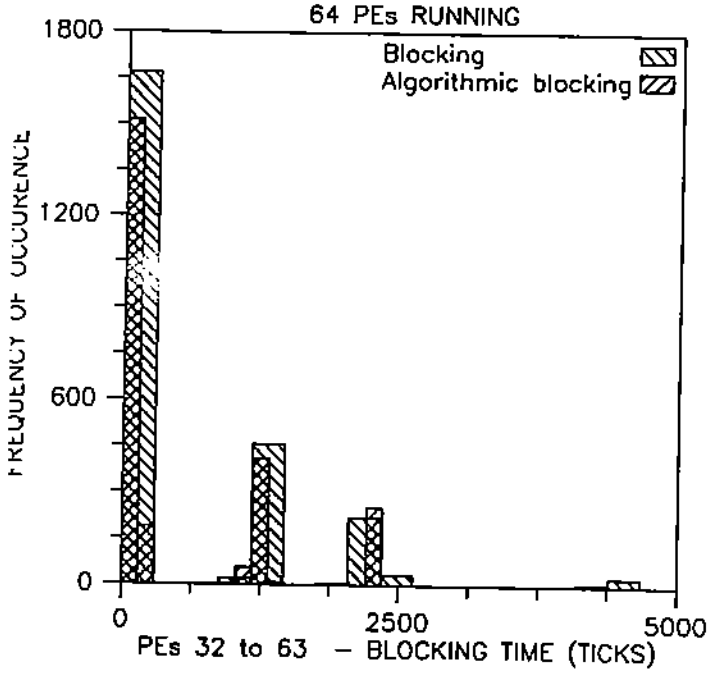
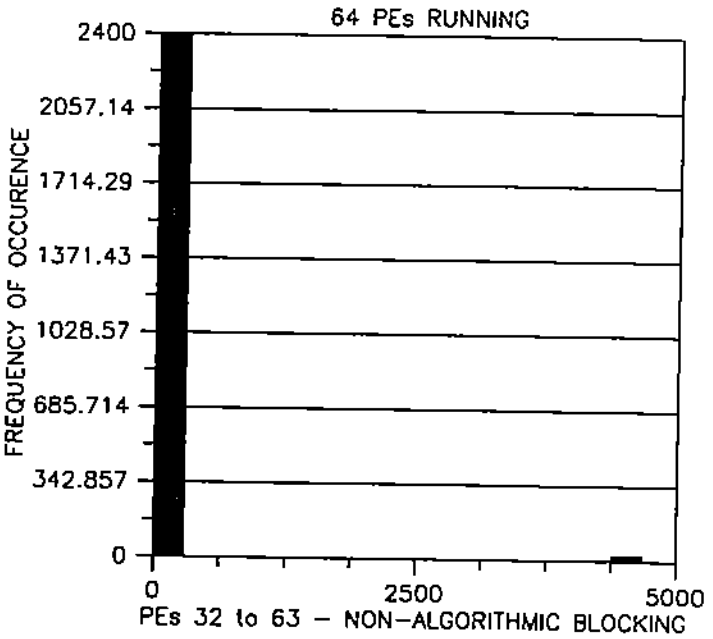
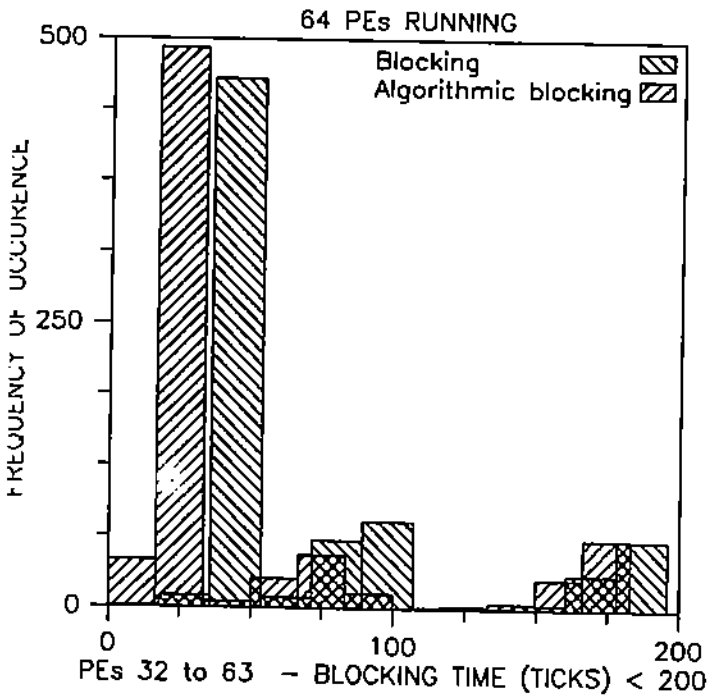


Figure 5. Histograms showing both total blocking and algorithmic blocking for PEs 8-15, (a) and (b), and for PEs 16-31, (c) and (d).







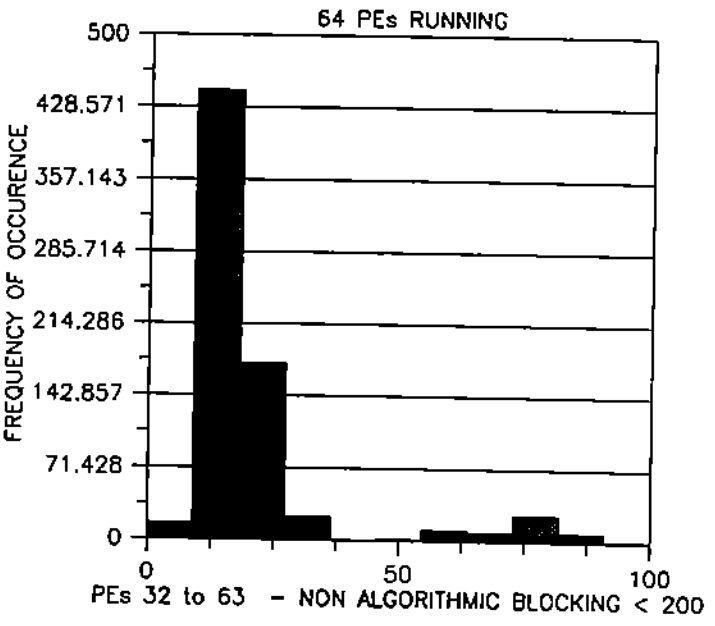


Figure 6. Histograms showing both total blocking and algorithmic blocking for PEs 32-63, (a) and (c), and non-algorithmic blocking for PEs 32-63, (b) and (d).