1992

Performance of Iterative Methods for Distributed Memory Processors

Dan C. Marintescu

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

E. Vavallis

Report Number:
90-979

PERFORMANCE OF ITERATIVE METHODS FOR DISTRIBUTED MEMORY PROCESSORS

Dan C. Marinescu
John R. Rice
E. Vavalis

CSD-TR-979
May 1990

PERFORMANCE OF ITERATIVE METHODS FOR DISTRIBUTED MEMORY MACHINES

D.C. Marinescu, J.R. Rice and E.A. Vavalis

Purdue University
Computer Science Department
West Lafayette, IN 47907

CSD-TR 979

December 16, 1992

Abstract

We consider iterative methods for the large linear systems arising from partial differential equation problems. These are somewhat banded but otherwise of no special structure beyond being sparse. We study the implementations and performance of several iterative methods on hypercube machines. We examine in detail the effects of communication and synchronization delays. Models are presented for these computations and projections made about performance for massively parallel machines. These models use the Events per Thread of Code (E/T) methodology of Marinescu and Rice and the performance is interpreted within this framework.

1 Introduction

Communication and control latency can strongly influence the performance of parallel computations on distributed memory multiprocessors. We have proposed a model of parallel computations, the E/T, Events per Thread of Control model, which defines a measure of the communication complexity. In this work we attempt to provide empirical evidence to support our model.

The E/T model describes a parallel computation $\mathcal{C}$ as a collection of P threads of control and E events. Informally a *thread of control* is an agent capable to perform some work in behalf of $\mathcal{C}$, and an *event* is an explicit action performed by a thread of control, in order to coordinate its activity with other threads of control. In a wider sense, an event is a change of state of a thread of control.

In the *E/T model*, a parallel computation $\mathcal{C}$ with P threads of control and E events, is described by its *characteristic function* g defined by $E = g(P)$. The model is based upon two assumptions:

(a) *Conservation of work.* Any work required by a computation $\mathcal{C}(1)$ with one thread of control, has to be performed by one of the threads of control of $\mathcal{C}(P)$, the parallel computation with P threads of control.

(b) $W(P)$, *the work required by a parallel computation is an increasing function of the number of threads of control, P.*

The first assumption needs little justification. It is an immediate consequence of the view that a thread of control is an agent performing some work in behalf of $\mathcal{C}$. To carry out a computation with P threads of control, simply means to redistribute in some fashion the work with otherwise would be carried out by only one thread. Call this constant amount of work reflecting the work conservation principle W_{cons}.

The second assumption is supported by the following arguments. An event is associated with every communication and control act. Any thread of control needs to communicate with other threads, at least at the instance when it is initiated when some work is assigned to it, and at the termination time, when it has to communicate its results. It follows that $g(P)$ is an increasing function of P. Moreover, any event requires a small amount of additional work, say θ, to be carried out by the thread of control when an event occurs. Let $W_{cc}(P)$ denote the additional amount of work required by $\mathcal{C}(P)$ for communication and control. The previous arguments show that $W_{cc}(P)$ given by

$$W_{cc}(P) \leq \theta \times E = \theta \times g(P)$$

is an increasing function of P. Thus, while $W_{cc}(P)$ might not increase monotonically, it is plausible to assume that the variations from the trend are small and that $W_{cc}(P)$ is increasing. But $W(P)$, the work carried by $\mathcal{C}(P)$ consists of at least two components, the first one W_{cons}, independent of P and the second one $W_{cc}(P)$, an increasing function of P

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

It should be pointed out that in practice, one cannot expect W_{cons} to be constant, but an increasing function of P, due to increasing algorithms overhead.

In this particular experiment, we investigate a particular algorithm we expect to have, a quadratic characteristic function $E = \mathcal{O}(P^2)$, because each thread of control has to communicate with all other threads once every iteration. A first objective is to show that the characteristic function is invariant to problem size.

2 Parallel Iterative Methods

In the framework of the //ELLPACK project [?], we are developing [?], [?] a portable general purpose library of parallel iterative methods for the solution of large PDE discretization systems of arbitrary PDE domain geometry. This library is mostly based on the sequential ITPACK [?], and is driven by the //ELLPACK interface tool [?]. In an effort to preserve all the robustness and all the convergence properties of the original sequential iteration schemes only rather minor modifications/additions where made on the original code.

Initially, using the //ELLPACK Domain Decomposer the PDE domain is partitioned, and each subdomain is assigned to a processor. The coefficients of the discretization equations local to each subdomain are reside on the associated processor which calculates the successive iterations of the local unknowns. Clearly, appropriate values of the unknowns on the interfaces of neighbor subdomains need to be communicated. It is important to point out that we assume that we do not have any specific information for the subdomains/processors mapping and thus we use *all-to-all* communication for appropriate values of the unknown vector.

In this study we will consider only the Chebychev iteration method with the Jacobi matrix as per conditioner, and for the rest of the paper we will refer to it as the **JACOBI SI** method. The JACOBI SI iteration scheme on each processor/subdomain can be described by

```
initializations;
scale_matrix_rhs;
```

```
bcast(interf);  \* everybody should have the initial *\
colaps(interf); \* solution of the interface unknowns *\
for (iter=1, iter<itmax, iter++) {
   if (not convergence) call jacobi_si;
}
unscale_matrix_rhs;
send_data_to_host;
quit;
```

We start by initializing various parameters, estimating the initial solution and scaling the matrix and the right hand side. A broadcast/colaps pair is used then for communicating the initial values of the interface unknowns. Then, a fast *barrier* synchronization mechanism is used to ensure that all processors enter *synchronized* the iteration loop.

The value of the logical parameter *convergence* is determined inside *jacobi_si* and is based on **global** convergence tests while the maximum number of iterations allowed *itmax* is set at the ELLPACK code level [?].

The pseudocode describing the computations performed by the routine *jacobi_si* follows:

```
compute_pseudo_residual;($,#) \* for stop/adaptivity tests *\
stop_adaptive_tests;($,#)
change_parameters;($,#)
compute_new_solution;(#)
bcast(interf);($)  \* everybody should have the updated
colaps(interf);($)   solution of the unknowns *\
```

The procedures marked with # involve computation while the ones marked with $ communication. The message sizes are either four bytes when calculating global inner products (change_parameters, stop_adaptive_tests and compute_pseudoresiduals) or are functions of either the local number of interface unknowns for symmetric matrices or the number of all local unknowns for non-symmetric ones. One can skip the *change_parameters* procedure by turning off the adaptivity at the ellpack code level. Notice that all communications are **all-to-all** using either a broadcast/collapse pair or bidirectional exchanges [?], and as such they almost synchronize all processors.

3 Performance Monitoring

This section describes the experiment to study the performance of iteration methods on a distributed memory system. The experiment uses the parallel ELLPACK (//ELLPACK) system developed at Purdue [?], running on a 128 processor NCUBE. The TRIPLEX tool set [?] is used to monitor the execution and to collect trace data.

The purpose of the experiment was to collect detailed information concerning the execution of a particular Single Program Multiple Data (SPMD) application, to study how this data relates to the high level characterization of parallelism in the framework of the E/T model [?], and to investigate how similar or dissimilar the behavior of the threads of control of an SPMD computation are.

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

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

The raw data were processed in several stages. First, the events outside the scope of Chebychev iterations were filtered out. Then a preprocessing to gather the data required by the E/T model was performed. The active time between events, the duration of an event (read/write) and the length of a blocking period, were obtained by correlating local events, events occurring in the same thread (on the same PE). The time for communication and control was computed as the difference between the duration of an event and the length of its blocking period. To compute the *algorithmic blocking* (defined as the interval from the instance a read is issued until the corresponding write takes place) it was necessary to correlate non-local events, events involving more than one thread. Finally, a statistical processing was performed in order to obtain data as described in Section 4.

4 Measurements

Two experiments were performed. The first one uses a rectangular domain and a 50×50 grid, and the second one an irregular domain and a 33×33 grid.

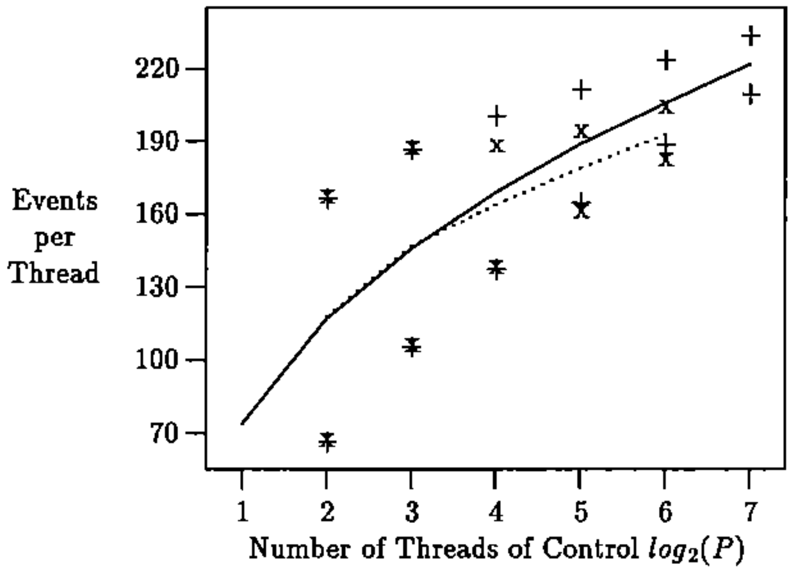

Figure 1: The expected number $g(P)/P$ of *events* per thread of control and a 95% confidence interval for it.

For each experiment, the PDE solver executes using a variable number of PEs, ranging from 2 to 128 for the first case and 4 to 64 for the second one. The actual decompositions corresponding to the 64 PE case are shown in Figures 2 and 3 for the 33×33 and 50×50 grid respectively. In the second case, the size of each subdomain and consequently the amount of computations performed by each PE is larger than the ones for the first case. The

Figure 2: The rectangular PDE domain and the decomposition obtained.

Figure 3: The non-rectangular PDE domain and the decomposition obtained.

number of interface points of each subdomain and consequently the amount of communication is also larger in the second case than in the first one. For all the graphs in this section we present data associated with the first problem with *solid* lines and + while for the second one we use *dashed* lines and x. The purpose of the experiments described in this paper is to obtain qualitative, rather than quantitative results. The two cases examined here show results in excellent agreement with one another.

Figure 1 presents the *characteristic function* $E = g(P)$, the number of units per thread of control function of the number of threads, on a logarithmic scale. As pointed out in [7], the characteristic function provides a signature of the algorithm and of its implementation. In this case $E = \mathcal{O}(P^2)$ as expected, since communication is done by broadcasting. Note that the algorithm itself requires that a subdomain communicates only with its neighbors, but there are two reasons why the implementation uses broadcasting rather than multicasting. First, the particular machine the experiment was carried on does not support efficient multicasting. Second, in order to multicast, each subdomain needs to know the ids of the PE to which its neighboring subdomains are assigned. But any algorithm to map dynamically logical subdomain ids to physical processors requires broadcasting, therefore it is unlikely that domain decomposition methods could be implemented with characteristic functions better than $E = \mathcal{O}(P^2)$. A common trend for all the measurements reported here is that the larger the number of threads of control, the smaller is the confidence interval for the quantities being measured.

In Figure 4, we show the timing of communication between two PEs, PE_i and PE_j. At time t_1, PE_i initiates a READ by calling a routine which starts to search for the desired data in the system buffer. If the data is not found then, at time t_2, PE_i becomes blocked waiting for the data. Later at t_3, processor PE_j initiates the WRITE which supplies the data requested by PE_i. At time t_4, the data begins to arrive at PE_i and at t_5 the WRITE operation terminates on PE_j. Finally, at time t_6, the READ terminates on processor PE_i when the data values are moved from the system buffer to the program and the computation proceeds. Note that *time* t_5 *not measured*, that is

PE_i measures t_1, t_2 and t_6,

PE_j measures t_3 and t_5.

Further, the times on PE_i and PE_j are measured by different clocks, so there might be some discrepancy in calculating the algorithmic blocking and the sum of propagation and data transmission times. The terminology

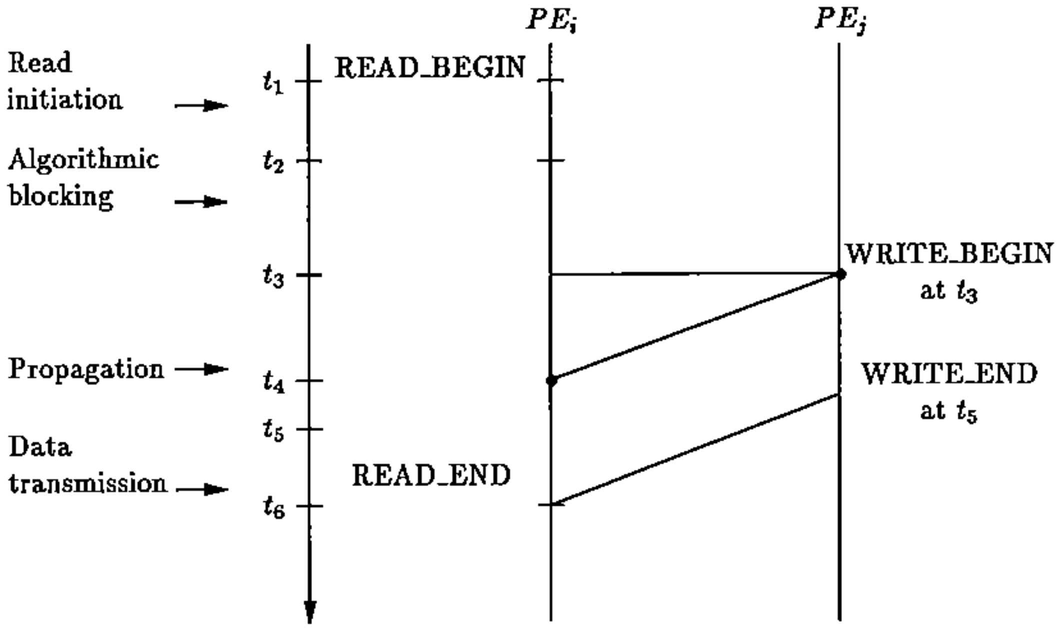

Figure 4: Communication involving blocking, the total blocked time for processor PE_i is the sum of the algorithmic blocking, the propagation delay, and data transmission time.

for times in this report is as follows (from Figure 4).

Active time:	Time between events
Read initiation:	$t_2 - t_1$
Algorithmic blocking:	$t_3 - t_2$ ($= 0$ if $t_3 < t_2$)
Propagation (not measured):	$t_4 - t_3$ ($= 0$ if $t_5 < t_2$)
Data Transmission (not measured):	$t_6 - t_4$
Non-blocked = Computing:	Active + Read initiation
Read:	$t_6 - t_1$
Write:	$t_5 - t_3$

Figures 5 and 6 show the expected *active period* per event and per thread respectively for the two cases. The active period is defined to be the time from the termination of an event to the beginning of the next event. An active period corresponds to the time a thread of control performs work assigned to it by virtue of conservation law. These figures confirm that the

50×50 grid assigns more work to each thread, for example, in case of a 16-way decomposition the expected active period per event is of about 45 ticks for a 50×50 grid as compared with less than 20 for the 33×33 grid.

Figures 7 and 8 present the *expected time for a single read* operation per event and the *total read time per thread* respectively, while Figures 9 and 10 present the same data for a write operation. A first observation is that the read time per event increases as P^2 in each of the two cases, while the write time per event is essentially constant. The average time for a write operation is of about 36 ticks for the first case and 12 ticks for the second one. As pointed out earlier (see Figures 2 and 3), the first case corresponds to large subdomains and a large number of interface points and the measurements confirm this.

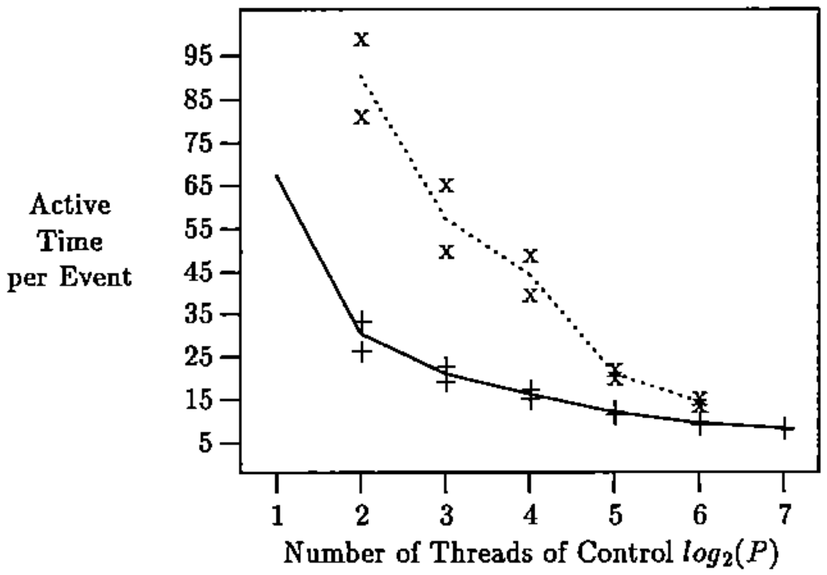

Figure 5: The expected *active* time between two consecutive events and a 95% confidence interval for it.

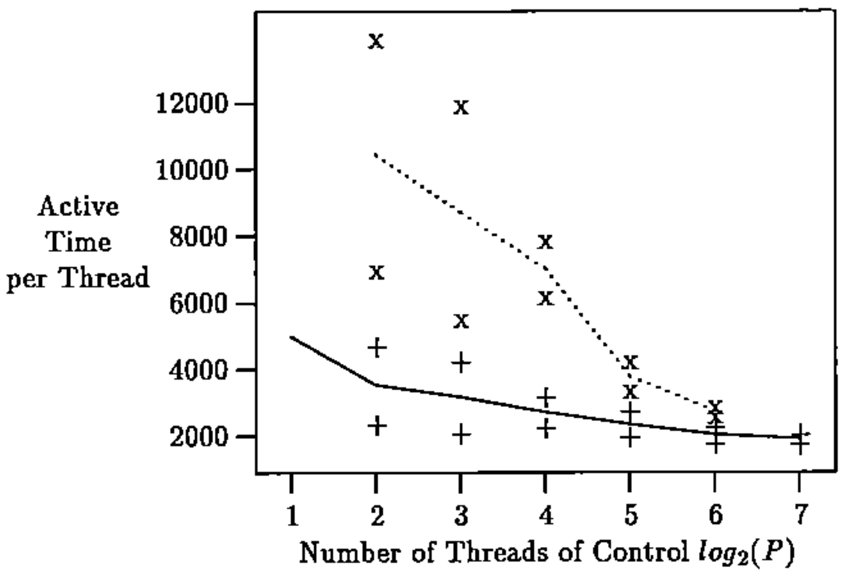

Figure 6: The expected *active* time for a thread of control and a 95% confidence interval for it.

It should also be pointed out that as the number of threads of control increases the number of write and read events increases logarithmically.

Here is important to notice that all figures associated with read and write events exhibit a rather strange behavior for 16 processors and that is more apparent when the size of the problem increases. Let us first consider the case associated with the write events since it is easier to analyze. Since the write operation is non-blocking, a PE simply copies the message into the system buffer, initiates the transmission and returns. Based on this, the expected write time is determined only by the size of the data being transmitted, which in turn depends upon the number of interface points of each subdomain. For simplicity let us assume a rectangular domain, a uniform global domain discretization and a chess-board uniform decomposition of it. In the case of two subdomains the total number of interface points for each subdomain equals to the global grid discretization lines in one direction. This number remains the same as the number of subdomains increases, in powers of two, up to 8 and is divided by two for 16 and there after. Based on the above observation the time for a write event is expected to be constant for up to 8 processors and then gradually drop while a similar behavior is expected for the read events. Nevertheless, the measured data indicate that other factors prevent the performance lines from dropping down. Communication interrupts, which have a high priority in NCUBE 1, slow down the write process. It has been observed [?], that as the message size decreases, the interrupt rate increases and the slowdown is more apparent. We believe this is the reason for the increase seen in Figure 7. The above obviously holds for the case of read events but here other more crucial factors (distance between communicating nodes and buffer managing) cause further performance degradation.

Let us now examine the 64-way decomposition. In the first case the expected read time is about 475 ticks, while in the second case it is of about 200 ticks. The measurements indicate that the number of read operations per thread equals the number of write operations. It follows that the expected duration of an I/O operation (read or write) is $\frac{475+36}{2} \cong 255$ ticks for the first case and $\frac{200+12}{2} = 106$ ticks for the second case. Consider now the experiment where 64 PEs are used. The expected active period for the two types of domain are 20 and 10 ticks respectively. It follows that the expected processor utilizations are

$$\eta_1 = \frac{20}{20+255} \cong 7.2\% \quad \text{and} \quad \eta_2 = \frac{10}{10+106} \cong 8.6\%.$$

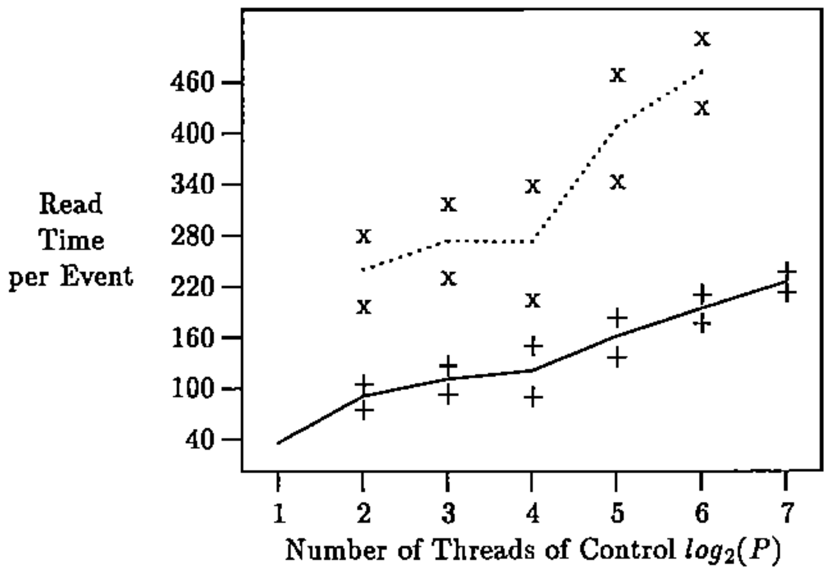

Figure 7: The expected time for a single *read* operation and a 95% confidence interval for it.

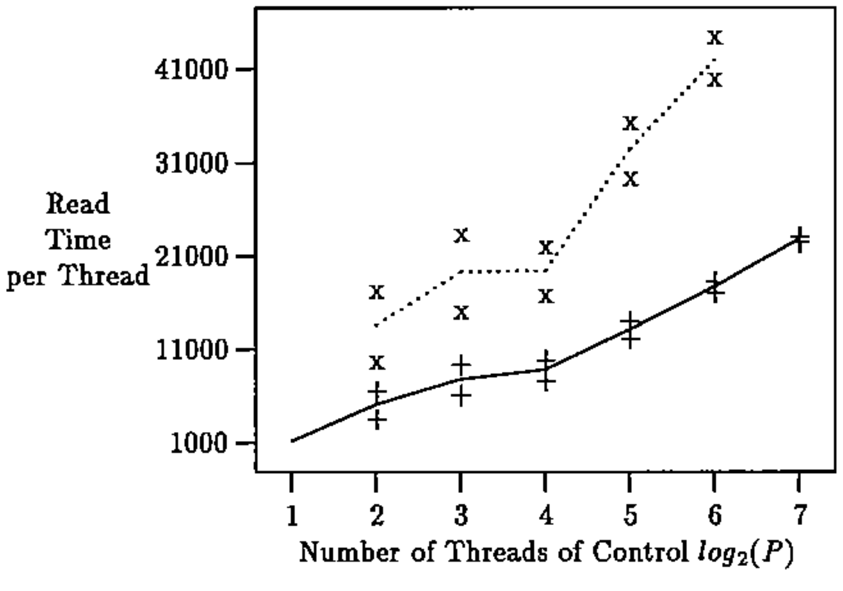

Figure 8: The expected time for all *read* operations of a thread of control and a 95% confidence interval for it.

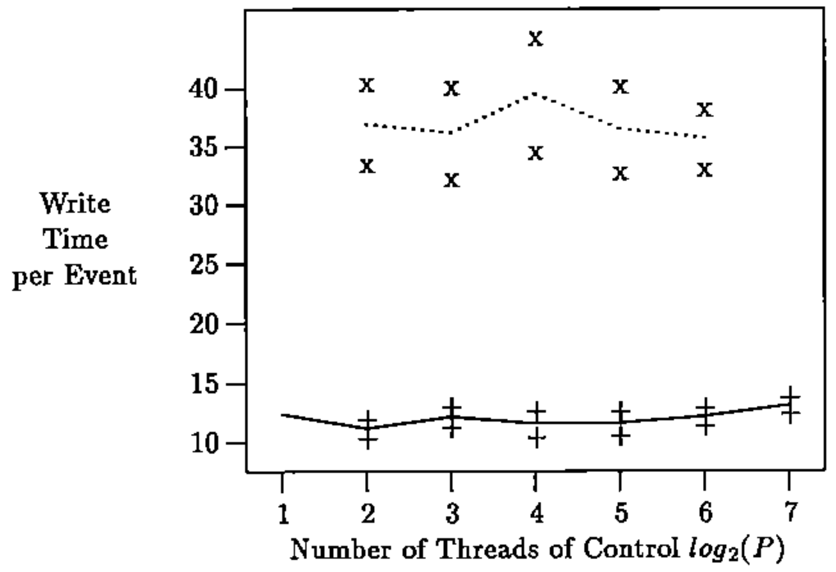

Figure 9: The expected time for a single *write* operation and a 95% confidence interval for it.

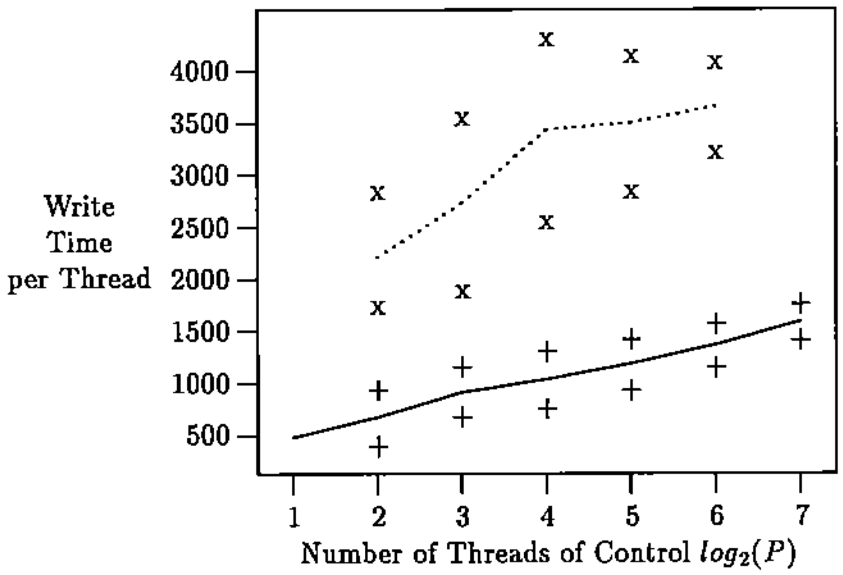

Figure 10: The expected time for all *write* operations of a thread of control and a 95% confidence interval for it.

Figures 11 and 12 show the expected blocking time per read operation and per thread of control respectively, for the two cases. Figures 13 and 14 show the blocking time as percentage of the total read time. They indicate that the blocking time increases as the number of threads of control increases, from about 70% in case of four threads to more than 90% in case of 64 threads. The blocking time is defined as the time from the moment a thread requests data and the moment the data becomes available (see Figure 10).

Since blocking is an important source of low processor utilization and consequently of low speedup, the blocking phenomena deserves to be scrutinized further. Figures 13 and 14 present the total blocked time as a fraction of the total read time and the total computing time. These percentages are very high, consistent with what we expect from the above discussions and data. These computations spend most of their time waiting for data, primarily because the communication is so slow compared to the arithmetic speeds. The principal cause of the waiting is the algorithmic blocking time. Figures 15 and 16 show that the total and individual lengths of these times grows rapidly with the number P of threads of control. Figures 17 and 18 show the fraction of the total blocked time that is algorithmic blocking. It is quite high and, comparing with Figures 13 and 14, one sees that the bulk of the time of a thread of control is spent in algorithmic blocking. This point is emphasized by Figures 19 and 20, which show the non-blocked or computing time behavior as the number of threads of control increases. Figure 19 shows the overall fraction of computing time, while Figure 20 shows that the expected fraction of computing time is equally low.

Figure 21 shows the minimum, average and maximum times for read operations per thread of control. The fact that the average is close to the maximum and far from the minimum indicates that there are a substantial number of long read operations compared to the number of short ones. Figure 22 shows the growth of the non-blocked time with the number P of threads of control. Since the active time is not growing (see Figure 23) and the number of events per thread is growing slowly with P, time for a transmission grows rapidly with P. This confirms the earlier discussion. Finally, Figure 23 shows that the active time per thread of control is decreasing as P increases, this is as one expects.

Further information about performance is obtained from Figures 1, 22, and 23. The active time per thread would decrease by a factor 64 as P goes from 2 to 128, if there were no additional "algorithmic overhead". Starting at 5000 (Figure 23), it actually drops only to 2000 instead of the expected value of 80 or so. Thus this computation has substantial overhead in the active computation beyond that due to the communication. We may also estimate the average transmission time per event, since it is the non-blocked time (Figure 22), less the action time (Figure 23), divided by the number of events (Figure 1). For $P = 2$, we see the average transmission time is (10,000-5,000)/70 = 70 and for $P = 128$ it is (1,600,000-2,000)/220 = 7,300. Thus the transmission time has increased by a factor of 100 (more than P) due to, we believe, the increased size of the cube (this should account for an increase of a factor of at least 6) and congestion on the communication paths.

Finally we note that the apparently erratic behavior that occurs for P = 2, 3, and 4 in many of the plots (e.g., Figures 6-9, 11-14, 17, 21, and 23), can be explained in terms of the underlying geometry of the PDE problems and it does not indicate random effects in the measurements.

5 Potential Program Improvements

The E/T performance analysis methodology has identified an important efficiency problem in this iteration algorithm. Since this is a typical example of a broad class of iterations that arise in many computations, it is appropriate to consider steps to alleviate this difficulty. We present observation of four types.

A. **Use Better or Less Synchronization.** We can look for less costly synchronization schemes. Since the simple one used is the best possible for true synchronization, we must sacrifice something. For example:

   A-1. *Synchronize only every 5 or 10 iterations.* This has an unknown effect on the numerical properties of the iteration, but it is certainly something to try.

   A-2. *Use adaptive synchronization.* Have every processor continue computing while the synchronization messages and analysis is taking place. Processor 0 initiates a true synchronization only when the timing of the messages indicates that the iterations have gotten too far from synchronized.

   A-3. *Local group synchronization.* Have small subgroups, changing over time, synchronize themselves. The numerical effects are unknown, but this is probably pretty safe.

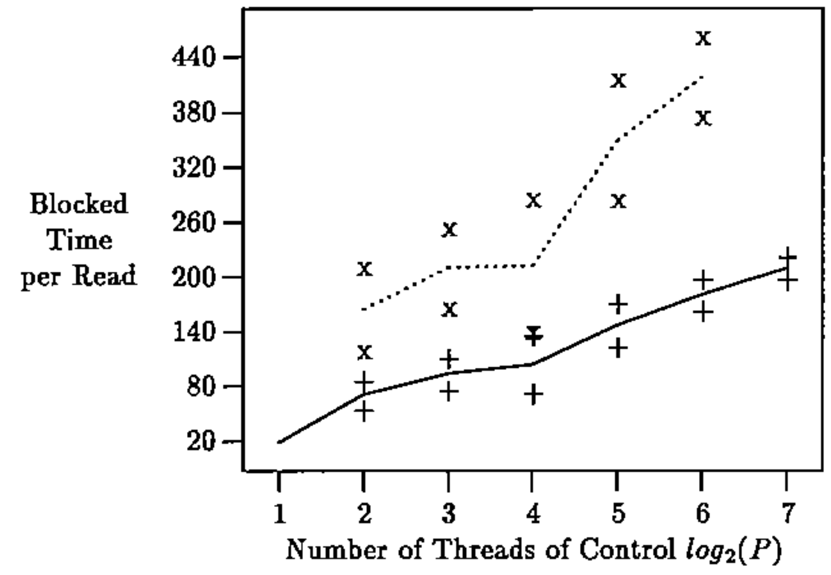

Figure 11: The expected time a thread is *blocked* during a read operation and a 95% confidence interval for it.

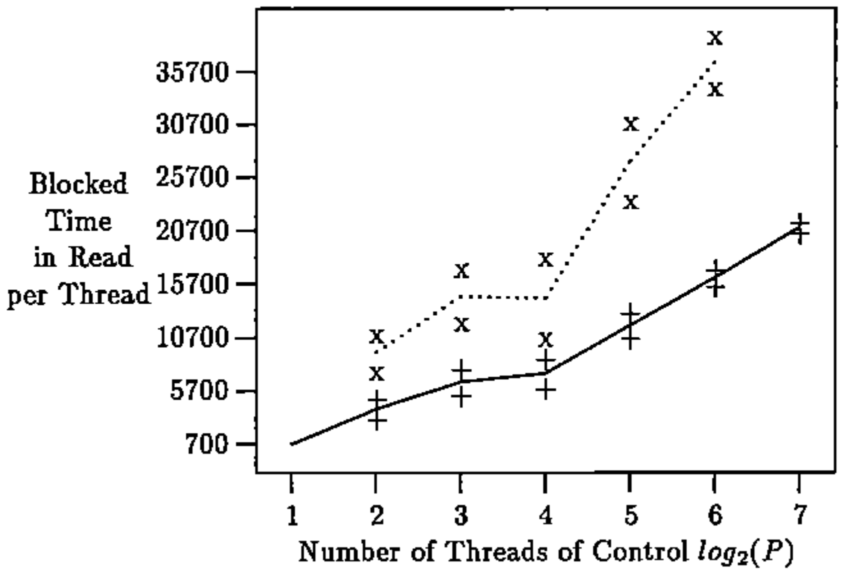

Figure 12: The expected time a thread is *blocked* during a read operation and a 95% confidence interval for it.

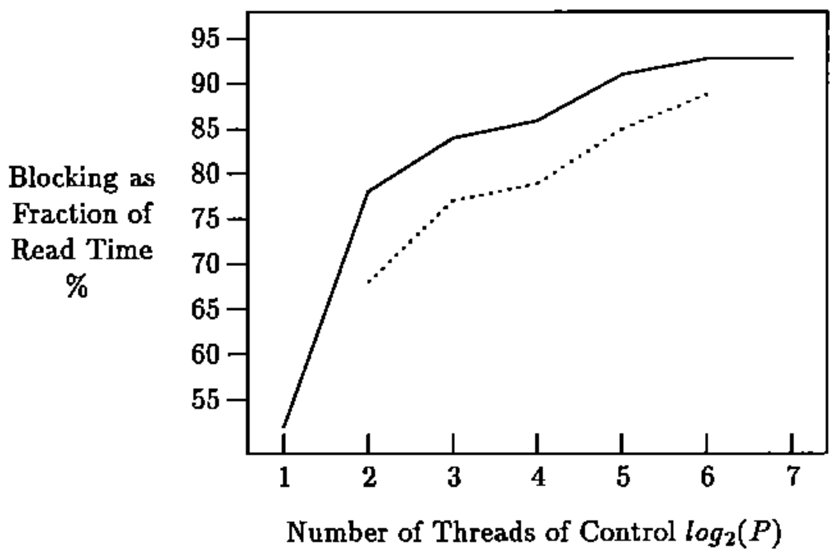

Figure 13: The expected *blocked* time during a read operation as fraction of the total read time.

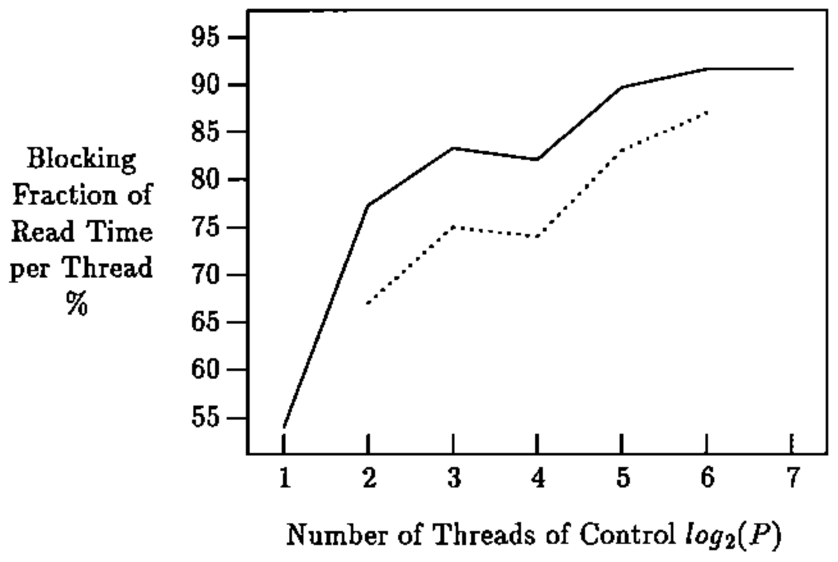

Figure 14: The expected *blocked* during read per thread as fraction of the total computing time per thread.

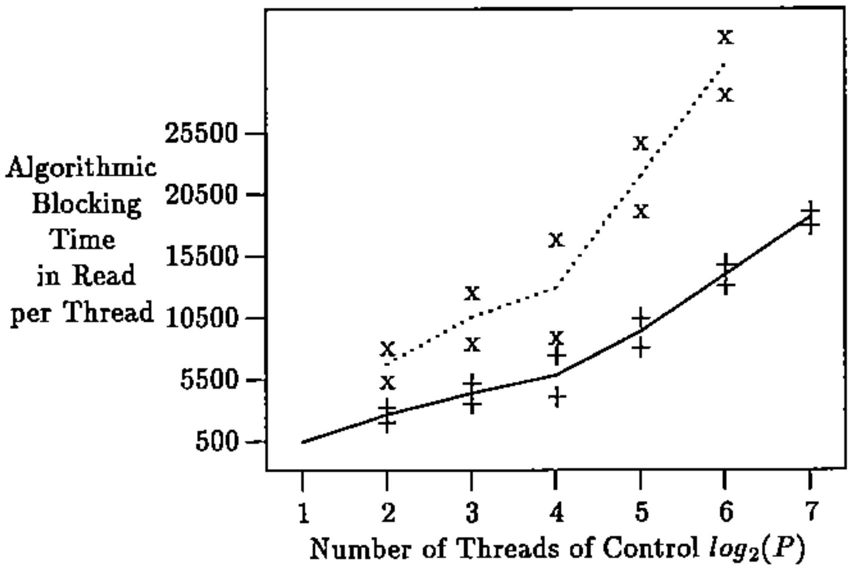

Figure 15: The expected *algorithmic blocking* time per thread and a 95% confidence interval for it.

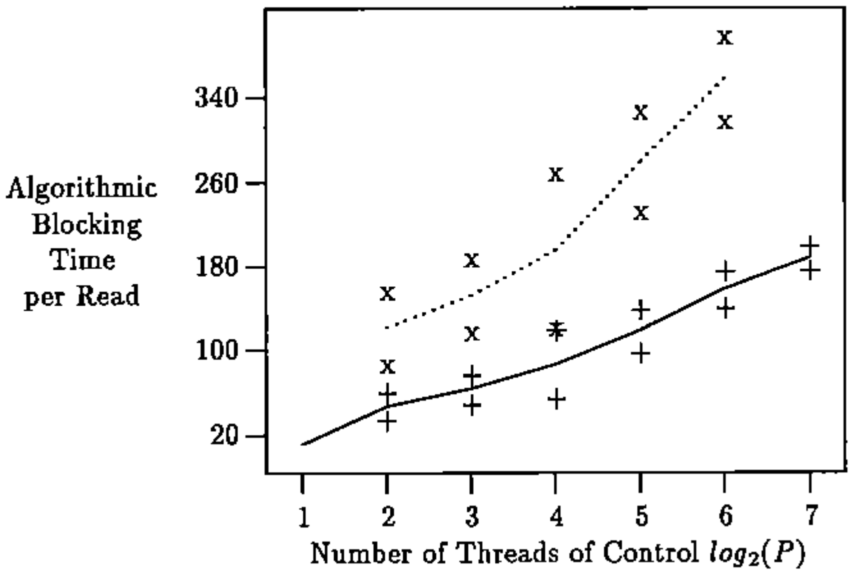

Figure 16: The expected *algorithmic blocking* time during a read operation and a 95% confidence interval for it.

B. **Special Load Balancing.** It is easy to see that processors that are on the first levels of the spanning tree, used for the broadcast/collapse pair, spend a long time waiting for the processors on the previous levels to finish. This fact could be taken into account when the equations are distributed to processors and thus the utilization could be improved. This probably would have a useful but not large effect on overall efficiency.

One could take another approach to detect an unbalanced load, that we might call the *local waiting time balance test.* Each processor measures how long it is blocked compared to its computation time. When it is blocked for too long it signals that the load balancing is bad and someone takes remedial action.

C. **Use Faster Communication Hardware.** The synchronization problem identified here is algorithmic in nature, the convergence theory of the iteration only applies if the iteration is synchronized. However, one can reasonably hope that exact synchronization is not really required for good convergence. If we had a perfectly balanced computation, it would be synchronized without explicit action. On the other hand, if the communication hardware were fast enough, we could synchronize without undue cost.

It is unclear whether communication hardware will keep up with speed improvements in arithmetic processors. The current use of network-like protocols for communication seems to make it impossible to have communication speeds comparable to arithmetic speeds. But then, such difficulties motivate people to devise better ways. As an indication of how things are going, one can compare the speeds of the NCUBE 1 and NCUBE 2 as below (the values given are approximate, all in microseconds).

	NCUBE 1	NCUBE 2
CPU cycle time	.14	.05
Add time	3.0	.35
Send 1 word to neighbor	450	140
Send 1 word across 128 cube	25000	152
Send 1000 words to neighbor	9000	5000
Send 1000 words across 128 cube	6×10^5	5000

D. **Increase Memory Per Node.** Existing hypercube machines tend to have too little memory per mode. Even iterative methods, which tend to have low memory requirements, are limited by the lack of memory. Consider an NCUBE 1 with 128 processors running this program.. The NCUBE 1 has 512 Kbytes of memory per node, or 128 Kwords. It is optimistic to expect that 60 Kwords are available for the linear system. In a reasonably compact sparse form one could hope to have about 5,000 equations per node (650,000 total). One iteration on 5,000 equations requires about 30,000 floating point operations. With 0.3 megaflops processors, the NCUBE 1 takes about 100 msecs (or 600 ticks) to do the iteration. Then it takes 1200 ticks to synchronize! The best utilization one could hope for is about 33%.

For the NCUBE 2 the speeds are increased by 10 for megaflops, 5 for communication and the memory is increased by 8 (but user memory probably increases by 12 or so). Thus we can have 60,000 equations in one processor's memory and an iteration takes about 160 msecs (or 1000 ticks). The communication time is decreased by a factor of 5 to give about 240 ticks for synchronization. Then the best computation/communication ratio changes from 600/1200 = 0.5 to 1000/240 = 4, a large but not overwhelming improvement. The best utilization one could hope for is thus about 80%. For computations which do not use all of a processor's memory, the ratios are smaller and the performance worse.

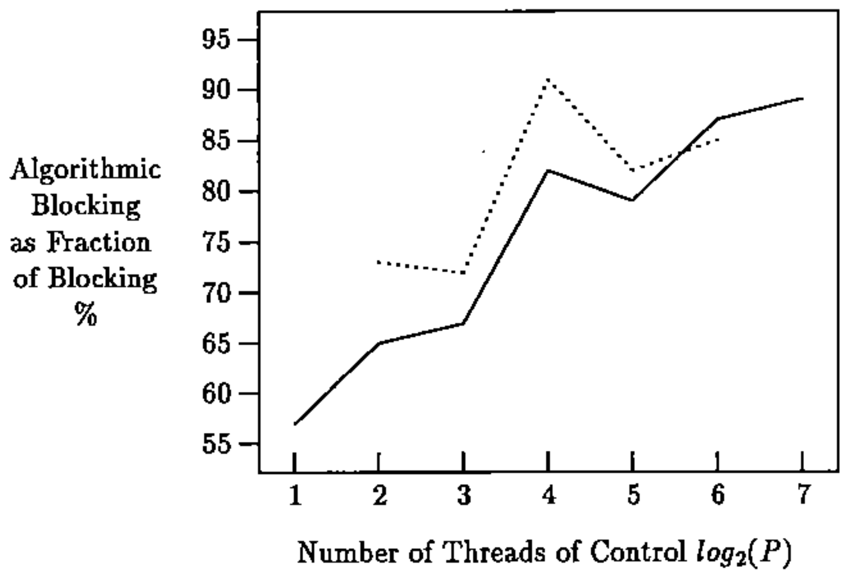

Figure 17: The expected *algorithmic blocking* time during a read operation as a fraction of the total blocking time during a read operation.

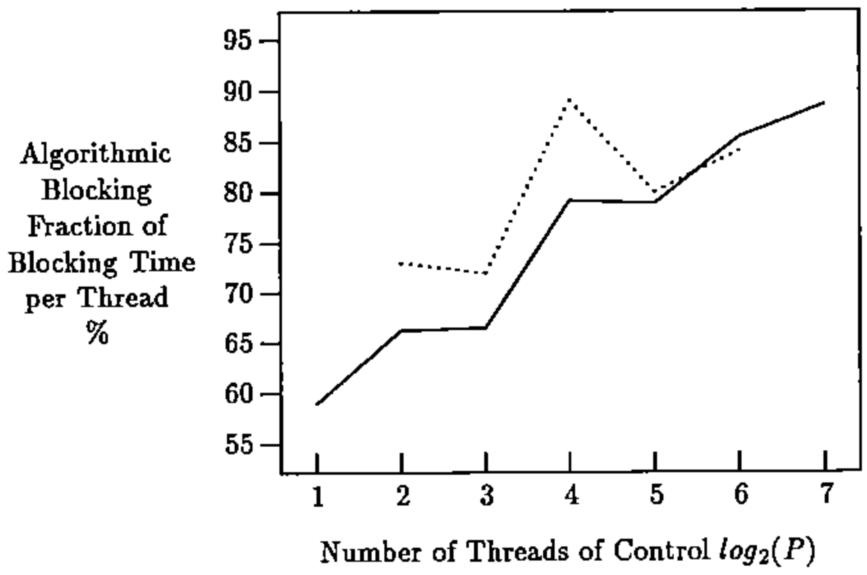

Figure 18: The expected *algorithmic blocking* time per thread fraction of the total blocking time during a read operation.

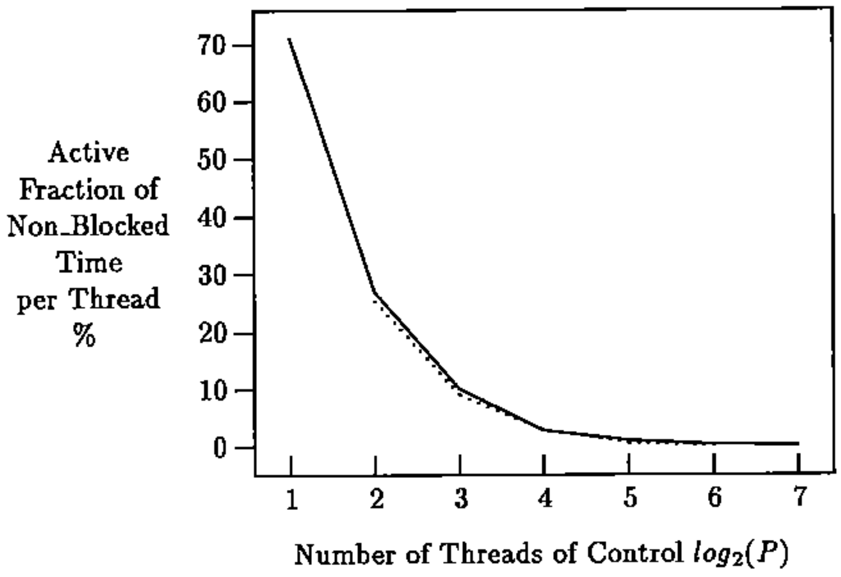

Figure 19: The expected *active* time as fraction of the computing time per thread.

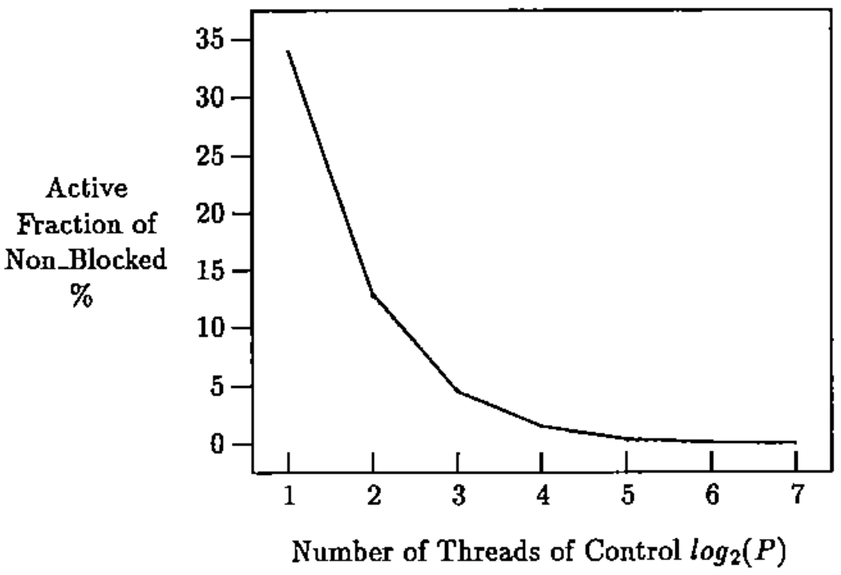

Figure 20: The expected *active* time fraction of the computing time between two consecutive events.

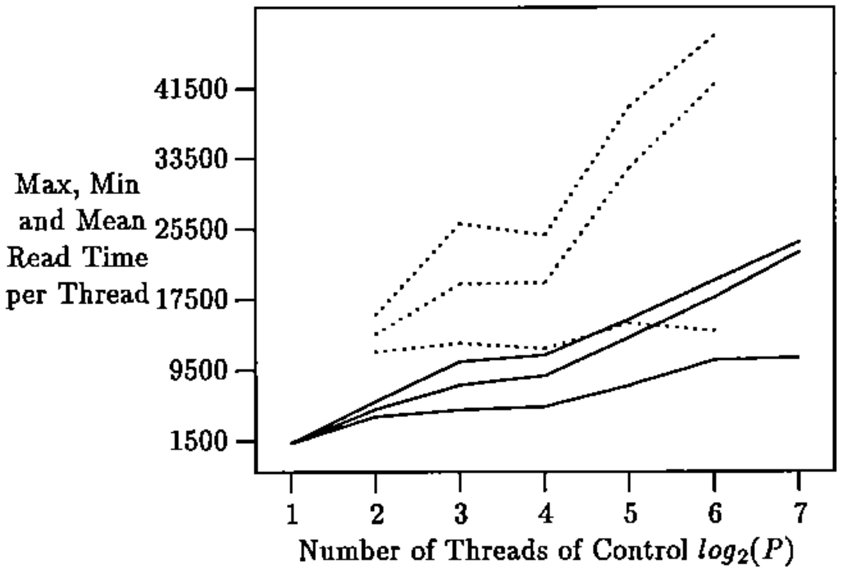

Figure 21: The minimum, the average and the maximum time for all *read* operations for a thread of control.

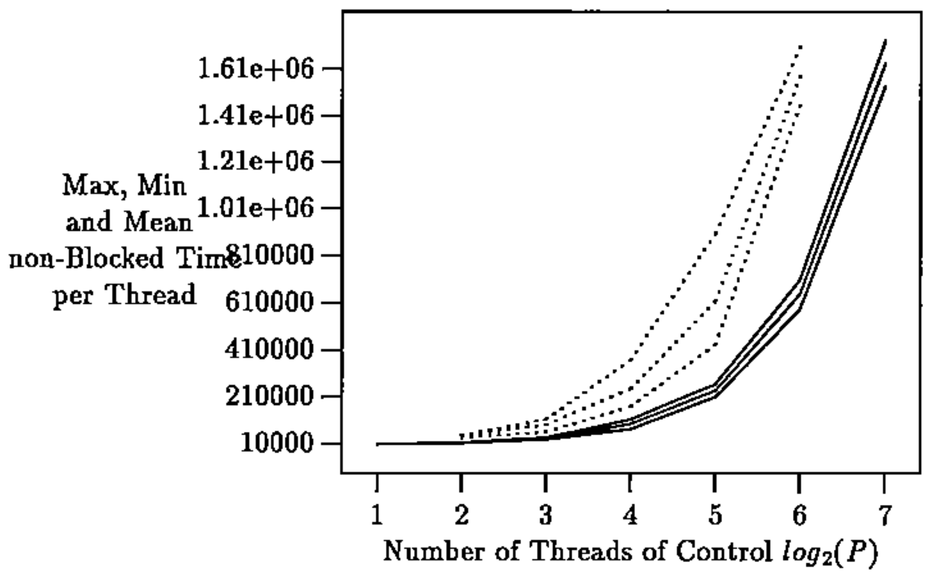

Figure 22: The minimum, the average and the maximum expected *non-blocked* time for a thread of control.

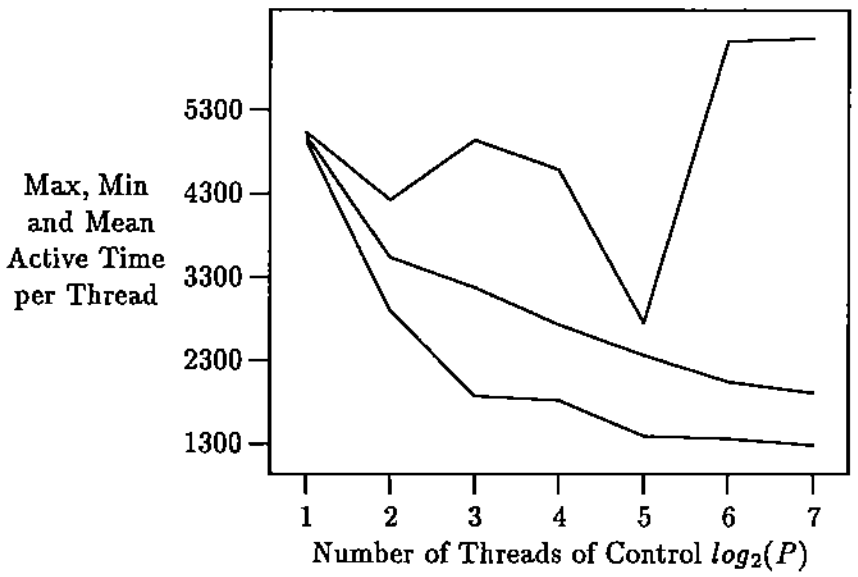

Figure 23: The minimum, the average and the maximum expected *active* time for a thread of control.