1991

Performance Experiments and Optimizations of PDE Sparse Solvers on Hypercubes,

Mo Mu

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

Jingwen Wang

Report Number:
91-045

PERFORMANCE EXPERIMENTS AND OPTIMIZATIONS OF PDE SPARSE SOLVERS ON HYPERCUBES

Mo Mu
John R. Rice
Jingwen Wang

CSD-TR-91-045
June 1991

Performance Experiments and Optimizations of PDE Sparse Solvers on Hypercubes

Mo Mu,
John R. Rice,
and
Jingwen Wang

Computer Sciences Department
Purdue University
Technical Report CSD-TR-91-045
CAPO Report CER-91-20
June, 1991

Abstract

In this report we present the results of experiments with the parallel sparse matrix solver of the Parallel Ellpack System. Three different hypercube parallel machines are used to compare and optimize its performance. After a brief description of the parallel sparse matrix solver and a presentation of the machine parameters and features, the measurements of performance of the sparse solver on three machines are compared. We observe that the performance of an algorithm is architecture dependent. This program achieves nearly perfect speed up on the NCUBE/2 and Intel iPSC/2 and it runs with disappointing speed up on the Intel iPSC/860 for small problems. Two bottle-necks of this inefficiency are located and a block-wrapping assignment and message merging method is devised to raise the computation granularity and reduce the communication overhead. Our experiments show that the method is very effective to improve the performance on the iPSC/860 when the problem size is small. The NCUBE/2 turns out to be the most balanced design for the problem. Despite the lower efficiency on the iPSC/860, the machine is seen to be far more powerful in terms of absolute speed.

I. Introduction

Parallel Sparse Matrix Solver is an algorithm for the direct solution of general sparse linear systems arising from discretizing partial differential equations (PDE) using a new organization of Gauss elimination [Mu and Rice, 1990a,b]. It is implemented as part of our Parallel Ellpack System [Houstis, Rice and Papatheodorou, 1989]. The purpose of this report is to investigate the performance of the sparse solver on different hypercube machines.

With the development of VLSI technology, faster processors are being used in distributed memory message passing (DMMP) parallel systems. The communication bandwidth of such systems, however, apparently has been lagging behind. This results in a decreased computation to communication ratio and thus decreased speed up. This is exactly the case with the iPSC/860, iPSC/2 and NCUBE/2. Programs running with nearly perfect efficiency on the

NCUBE/2 or iPSC/2 machines may be very inefficient on the new iPSC/860. New approaches to improve performance therefore are needed to make parallel processing practical on new generations of machines. One way is to upgrade the communication hardware, but that is beyond the control of algorithm designers. Another way is to further optimize the algorithms. Designing efficient software is a challenge for the new generations of DMMP multiprocessors.

In this report, we evaluate our sparse matrix solver on the NCUBE/2, iPSC/2 and iPSC/860. Performance data are measured and studied. Based on the observation of the performance data, two bottle-necks in the original algorithm are located and removed. A block-wrapping assignment and message merging technique are used to raise the computation granularity and reduce the frequency of communication. The experiments show that this technique is very effective in improving the performance when the problem size is small. It is also observed that despite the lower efficiency on the new iPSC/860, the absolute speed of this machine is substantially higher than for other machines.

II. Parallel Sparse Matrix Solution Model

We start with a brief description of the computation. A geometric approach to develop parallel sparse solvers for PDE problems suggested by [Mu and Rice,1989] is employed in the solver. It was shown to be particularly efficient for linear systems arising from solving partial differential equations using domain decomposition with a nested dissection ordering [Mu and Rice, 1990a].

The parallel sparse solver implemented in the Parallel Ellpack System uses the concept of **Elimination Tree** to represent data dependencies and parallelism. For simplicity of description, we consider PDE problems on a rectangular domain Ω, although the approach can be extended easily to general domains. Suppose we have $p\,(=2^{2d})$ processors available, 2^d in each direction. By domain decomposition Ω is divided into p subdomains Ω_{ij}, $i, j = 1, 2, \ldots, p^{1/2}$ as shown in Figure 1. One puts a local grid on each subdomain Ω_{ij} and discretizes the local problem whose solution U_{ij} only depends on unknowns at grid points of $\partial\Omega_{ij}$, the boundary of Ω_{ij}.

Initially, all interior unknowns U_{ij} are eliminated locally as in the standard domain decomposition approach. This step is obviously totally parallel. Then, all processors participate in eliminating interface unknowns. Communication and coordination among processors are needed during this process. To exploit more parallelism, we use dissection in alternating directions to partition the interface set into several levels suitable for a hypercube machine, each level consists of several **separators**, groups of unknowns which separate regions. The partition, which we call the **incomplete nested dissection decomposition**, is shown by Figure 2 with circles representing the unknowns interior to the subdomain Ω_{ij}, the boxes representing the **separators**. For simplicity, they are all called **subdomains** of this domain decomposition in the nested dissection manner and are numbered from top level to bottom level as shown in Figure 2.

This domain decomposition naturally inherits certain parallelism because the PDE discretization process leads to a local or boundary dependence property for interfaces. For example, if we consider the union of subdomains 16, 17, 8 as a more general subdomain Ω'_8 then the local interior solution set U'_8 is uniquely determined by unknowns on $\partial\Omega'_8$. This relation holds similarly for groups at higher levels of the dissection decomposition for the unknowns

arising in PDE applications. This local dependency can thus be described by a binary tree or **Block Elimination Tree** as shown in Figure 3 with each tree node corresponding to a subdomain in the decomposition.

The assignment of unknowns to processors includes both the assignment of the problem data and the computation associated with the data. Assume that the number of processors used is equal to the number of nodes on leaf level 0. First, the root node of the elimination tree is assigned to the whole hypercube and then the hypercube is split into two subcubes to which the two descendent subtrees are assigned. This process goes on recursively until all subtrees become assigned to single processors. The assignment within each node is also potentially arbitrary. This assignment is termed **subtree-subcube** assignment [George and Liu, 1987].

Ω_{11}	Ω_{12}	Ω_{13}	Ω_{14}
Ω_{21}	Ω_{22}	Ω_{23}	Ω_{24}
Ω_{31}	Ω_{32}	Ω_{33}	Ω_{34}
Ω_{41}	Ω_{42}	Ω_{43}	Ω_{44}

Figure 1. Domain decomposition of a rectangle.

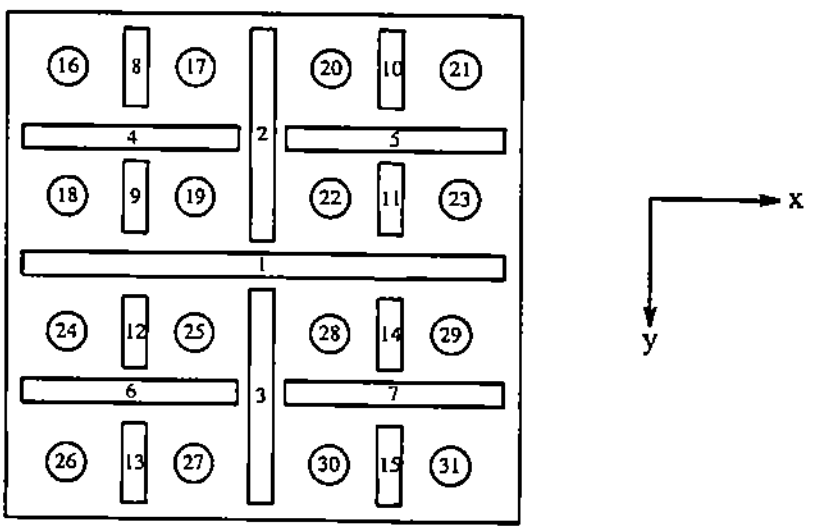

Figure 2. Incomplete nested dissection of the domain decomposition of Figure 1.

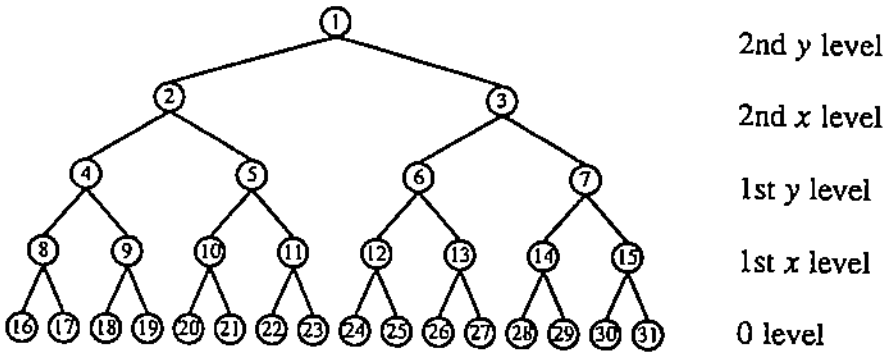

Figure 3. Block elimination tree based on the incomplete nested dissection of Figure 2.

A **grid-based subtree-subcube assignment** was proposed by [Mu and Rice, 1991] to reduce the redundant dependencies in the subtree-subcube assignment. For instance, in Figure 2, the elimination on separator 4 involves the coordination of 4 processors (corresponding to interior subdomains 16,17,18,19). Further, we see that subdomain 16 need not coordinate with 19, and subdomain 17 need not coordinate with 18 because there are actually no local data dependencies between them. We therefore split the separator 4 into 2 segments so that each segment only involves 2 processors. In the work presented here, we are using this grid-based subtree-subcube assignment.

A new organization of the Gauss elimination and a mixed type of data structure are used in the sparse matrix solver [Mu and Rice, 1990b] to reduce the heavy communication overhead in the early stages of the elimination and the data structure manipulation in the later stages of the elimination. The new organization of the parallel Gauss elimination is described as follows. First, each processor independently performs the standard Gauss elimination on its subdomain equations in its leaf node. Second, each processor independently computes its local "**B2**" matrix by a back solve process, sends **B2** to the related processors, and then uses **B2** itself to compute the **Schur complement** D (the interface submatrix). Third, each processor receives the **B2** matrices from other processors and uses them to compute D. Finally and fourth, all processors participate in factorizing the Schur complement D using a fan-out parallel sparse solver which is basically a process of alternatingly performing *local* and *global eliminations* along the elimination tree. Therefore, the major interprocessor communication work occurs in the fourth step where the interface submatrix D is factored, while the sparse data structure manipulation is required only in the early stages where the subdomain unknowns of the leaf tree nodes are eliminated.

III. The Machine Parameters

This section presents some basic performance data about the hypercube machines used, the NCUBE/2, Intel iPSC/2 and Intel iPSC/860. Some of these data can be found in [Bomans and Roose,1989] and [Dunigan,1990]. We compare these machines in terms of relative processor speeds, communication rates, communication start-up times, clock rates, and memories. The processor speeds are in units of a single NCUBE/2 processor power, measured in terms of our sequential sparse matrix solver code. Table 1 lists these values.

Table 1. Basic parameters of the NCUBE/2, iPSC/2 and iPSC/860. The node speed is relative to the NCUBE/2 processor and the range of possible memory sizes per node is given.

Machine	Processor	Clock rate	Comm. rate	Comm. Start-up	Node speed	Memory/node
NCUBE/2	Custom 64bit	20MHz	2.22MB/s	160 μs	1	1 → 64MB
iPSC/2	80386/387	16MHz	2.8MB/s	390 (700) μs	0.674	1 → 16MB
iPSC/860	i860	40MHz	2.8MB/s	75 (136) μs	9.78	1 → 16MB

One thing worth noting is that the Intel machines have very different communication start-up times for short messages and long messages. The numbers in parentheses of Table 1 are the start-up times for messages longer than 100 bytes. This difference is caused by the different communication protocols used by the Intel machines for long messages and for short messages. For messages longer than 100 bytes, 2 handshaking messages between the source and destination nodes are inserted automatically by the system to insure that there is enough space in the destination to buffer the message. This has made the machines less efficient in exchanging long messages. The minimum time to exchange a message on the iPSC/2 can easily exceed a millisecond.

The NCUBE/2 and iPSC/2 have roughly equal processor power and communication rates, but the NCUBE/2 has a much smaller communication overhead per message than the iPSC/2. Both the iPSC/2 and iPSC/860 use the same communication hardware. The iPSC/860 uses far more powerful processors than the other two machines. In contrast, it has a much worse computation to communication ratio and tends to be much less efficient for parallel programs.

All three machines have big address spaces. However, the iPSC/2 nodes are usually equipped with only 4MB of memory. Our NCUBE/2 has 4MB of memory for each of the first 16 processors and 1MB of memory for the other 48 processors. Our iPSC/860 has 16MB of memory per node.

The iPSC/860 provides many additional communication calls to the user, such as global collection, global product and sum, multicast, etc. However, our experiments show that only the broadcast communication call (implemented by **csend** or **isend** call by setting the destination node index = −1) reduces the communication time effectively. All other calls have about the same performance as if they are implemented by the user in loops of simple communication calls.

IV. Basic Experiments

We test the performance of the sparse matrix solver on these three machines using 16 processors. The problem we consider is the two dimensional Poisson equation with Dirichlet boundary condition on a rectangular domain. We only measure the LU factorization performance because there are, as yet, no efficient parallel methods for back substitution. The curves of speed up versus problem size are plotted in Figure 4. The horizontal axis represents the number of grid points along one of the two dimensions and we always assume that the numbers of grid points are the same in both dimensions.

We see that with the same program, the NCUBE/2 shows very good speed up and approaches perfect speed up (16) very quickly. The iPSC/2 also shows good speed up but is worse than the NCUBE/2. The speed up on the iPSC/860 is disappointing when the problem

size is small. In the worst case shown, there is almost no performance gain by using the multiprocessors (the speed up is close to 1).

One apparent and dominating reason for the poor speed up on the iPSC/860 is the big difference in machine parameters. The i860 processor has nearly 10 times the computing power of a NCUBE/2 processor, but the communication performance has little improvement. The speed up on the iPSC/2 is also poorer than on the NCUBE/2. In fact, the communication performance of iPSC/2 is actually worse than the NCUBE/2 because of the higher overhead per message. The NCUBE/2 appears to be the most balanced machine design for our problem.

It is also of interest to know the actual processing speeds of these machines. Figure 5 shows the parallel execution speeds of the sparse solver on the three machines. Again, the speeds are in units of the power of a single NCUBE/2 processor measured in terms of our sequential sparse code, i.e.

$$S_p = \frac{t_s}{t_p}$$

where S_p is the parallel execution speed, t_s is the sequential time running on the NCUBE/2 node and t_p is the parallel time running on the given multiprocessor. We can see that despite the lower efficiency of the iPSC/860, the actual processing speed is extremely attractive for reasonably large problem sizes, the other two machines are not comparable to it.

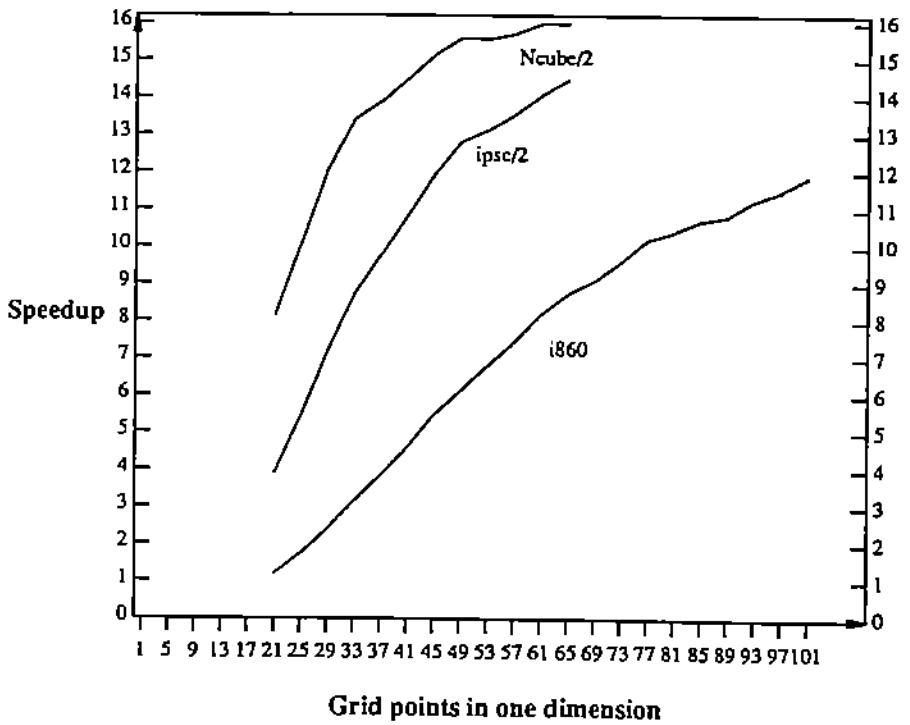

Figure 4. Speed up vs. problem size for a Poisson problem solution using the Parallel Sparse Matrix Solver on three hypercube machines with 16 processors.

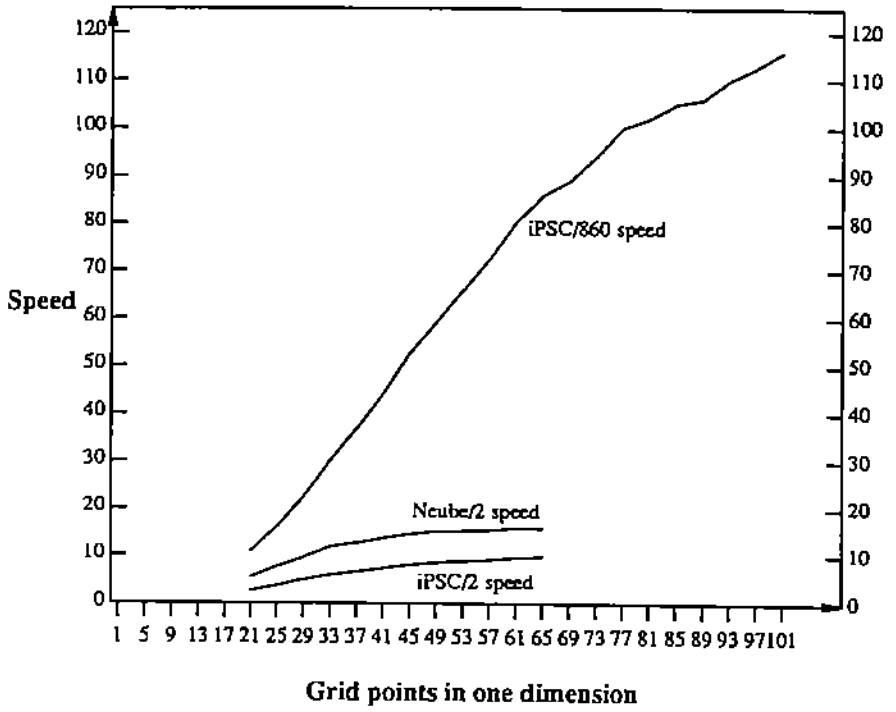

Figure 5. Parallel execution speeds of hypercubes relative to a single NCUBE/2 processor on the computation of Figure 4.

V. Algorithm Optimizations

It is seen that when the size of the problem is small, the speed up on the iPSC/860 is very disappointing. It is therefore very desirable to optimize our algorithm to achieve a reasonably good speed up for small problem sizes.

By detailed timing on the program execution, we found that the bottle-neck lies in the frequent communications during the eliminations of the unknowns in the separator domains. Since these unknowns are global to a group of processors, the corresponding pivot equations from a processor need to be sent to several processors. Thus multicasting is required. There is nothing one can do about these requirements because they are determined by the data dependencies of the problem. However, there is a chance to reduce the communication frequency by merging several messages together.

In the original algorithm we assigned the unknowns in a segment of a separator domain to the associated processors in a wrapping manner. Such an assignment helps make the computation load on a segment evenly distributed among the processors. When a processor holds a pivot equation during the elimination of an unknown in the segment, it needs to send the equations to all the involved processors.

Because the communication on the iPSC/860 is much slower compared with the computing power of the processor, it takes very little time for a processor to perform the elimination on an unknown. Thus the behavior of each processor is the repetition of a short burst of computing accompanied by a long communication time. When the processor finishes all the local updating associated with a pivot it holds, the communication to send out the pivot equation

still has a long time to go. Since the next unknown is assigned to another processor by wrapping, it usually has to wait for a long time before it can proceed. The waiting on these synchronization points is very frequent. Furthermore, waiting times at these points can be significant because the communication is often multicasting, which usually takes several milliseconds.

Based on the above observation, we have changed our algorithm in the following way. Instead of assigning the segment unknowns on a point wrapping basis, we assign a block of consecutive unknowns to each processor involved in the segment, also in a wrapping manner. This is called **block wrapping** while the original method is called **point wrapping**. In this way, a processor can do a sustained computation over a block of unknowns with each communication. Each message now holds several pivots instead of only one. This is the common practice in the dense matrix computations in order to both raise the computation granularity and to reduce the communication frequency [Geist and Heath, 1986], and has potential of performance improvement in this situation. A similar idea [Zmijewski, 1989] has been mentioned for sparse Cholesky factorization. Because the size of the message is small for the small problems where we are trying to optimize, it takes almost the same time to send the merged message corresponding to several pivots as the original message corresponding to only one pivot equation.

One negative point of this method is that it usually delays a processor starting to send the pivot equations because now a processor has to merge several pivots together. This conflicts with the pipelining idea of sending messages out as soon as they are available for other processors to use. However, since the computing speed is much faster than communication, the benefits we receive from reducing pipelining may well outweigh the loss. A second drawback is that the computation load on a segment may not be as balanced as with the point-wrapping method. In the extreme case, the size of the wrapping block is large enough so that all unknowns on a particular segment are within one block and can thus be assigned to a single processor leaving all other processors idle. In this case, the parallelism in the computation of this segment would be 1. Therefore one needs to choose an optimal wrapping block size for each problem. Different block sizes will achieve a different speed up. Figure 6 shows the relationship between wrapping block size and speed up for several sample problem sizes on the iPSC/860.

The first conclusion we can draw from Figure 6 is that there is obvious performance improvement from using the block wrapping assignment and message merging approach. The speed up of the 25×25 problem has been raised from 1.72 to 4.69 with a block size of 6. The second conclusion is that this method is less effective for bigger problem sizes as the curves become flatter and flatter when the problem size grows. The third conclusion is that there is an optimum wrapping block size for each problem size.

We have collected the results of our experiments on the iPSC/860 with the block wrapping method for various problem sizes. Several sample speed up curves are shown in Figure 7, where sp_i represents the curve for block size i. The performance is clearly improved when the problem size is small. We can still see some improvement for problem sizes up to 93×93. This method is less effective for large problem sizes. The reason for this may be two-fold. First, when the problem size is very large, there will be little effect by merging two or more messages into one because now the message size is already quite large and the time it takes to transfer the message is almost proportional to the message size. Thus the only savings by merging would be the communication start-up time, which is now only a very small fraction of

the total communication time. Second, the longer delay spent in waiting for and packing enough messages together may even make the performance worse. In fact, for large problems, the previously mentioned bottle-neck no longer exists. The real bottle-neck now is the large volumes of data exchanges and not the frequency of communications.

The results in Figures 6 and 7 indicate that no block size is optimal for all problem sizes. It is possible to use different wrapping block sizes for different problem sizes to achieve the best performance. Figure 8 gives the speed up of an optimal selection of block sizes for various problem sizes.

We also report on experiments made to see if the above method helps for the iPSC/2 machine. The results for block sizes 2 and 4 are plotted in Figure 9. There is some appreciable improvement when the block size is 2 and the problem size is small. When block size is 4, the speed up is improved a little when the problem size is very small and soon becomes worse when the problem size grows beyond 37×37. This shows that on the iPSC/2, the above mentioned bottle-neck is much less severe than on the iPSC/860.

The speed up we obtain on the hypercube machines is very satisfactory for such problems. We have achieved a much better speed up on the iPSC/2 than the results for similar problems reported earlier [Ashcraft et al., 1990], although there exist some subtle differences in the problems (their problem was the Cholesky factorization of sparse symmetric matrices) and the machines (they used the SX iPSC/2 node with a fast scalar processor).

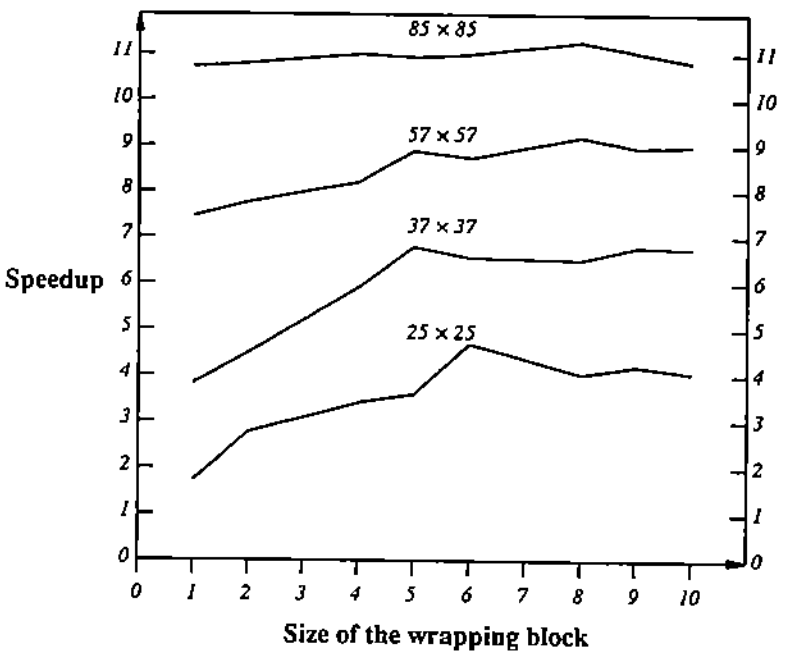

Figure 6. The effect of wrapping block size on the speed up achieved on the iPSC/860 using 16 processors.

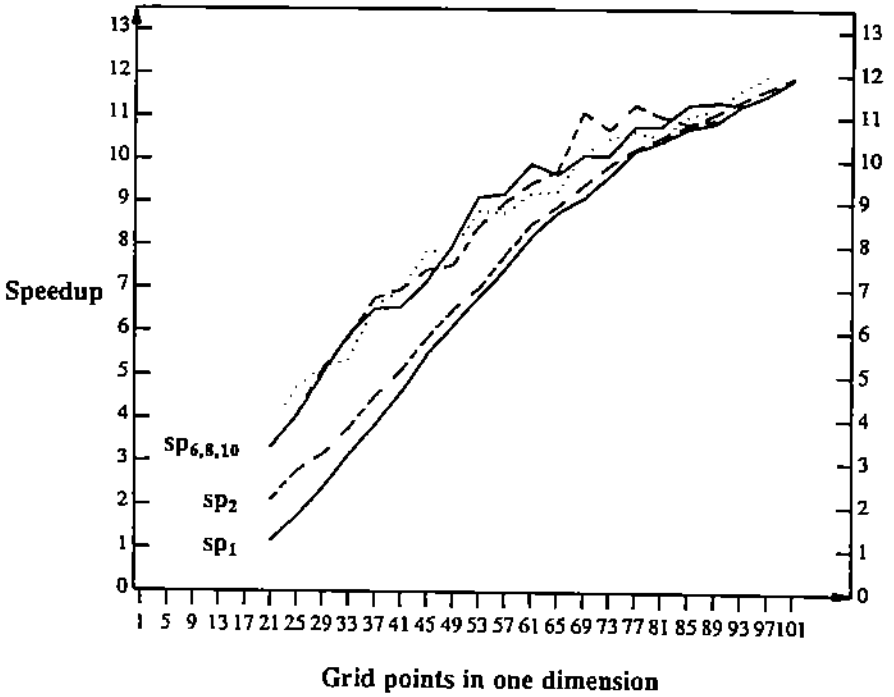

Figure 7. A second view of the effect of block wrapping size on speed up achieved on the iPSC/860 using 16 processors. The lower, solid curve is the standard wrapping (no blocking) and the higher curves are for wrapping block sizes of 2, 6, 8 and 10.

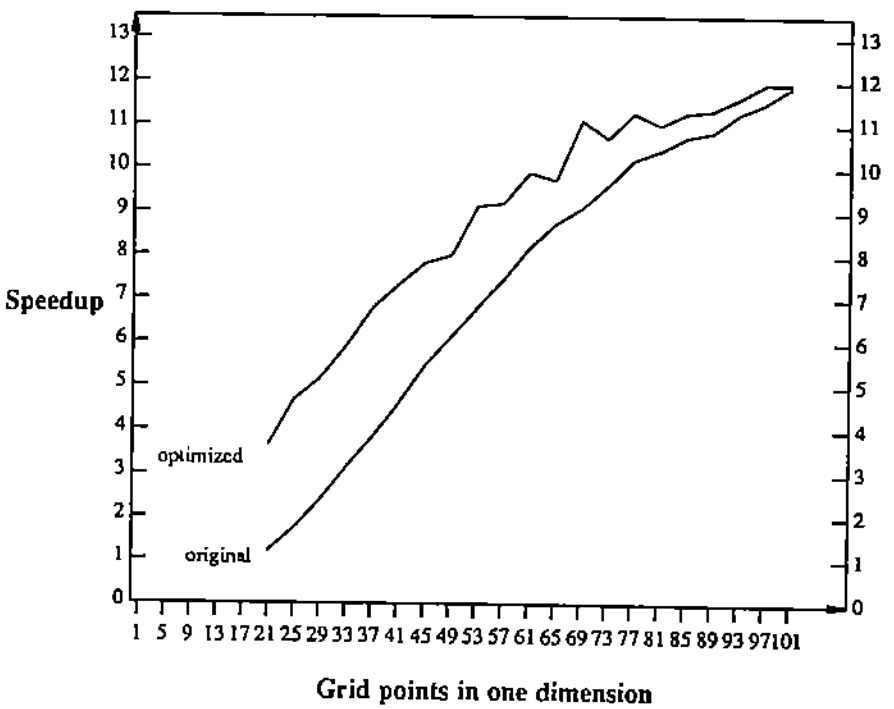

Figure 8. Performance of optimized wrapping block sizes. The standard wrapping (no blocking) is shown (bottom) with the best wrapping block size (which varies with the grid dimension).

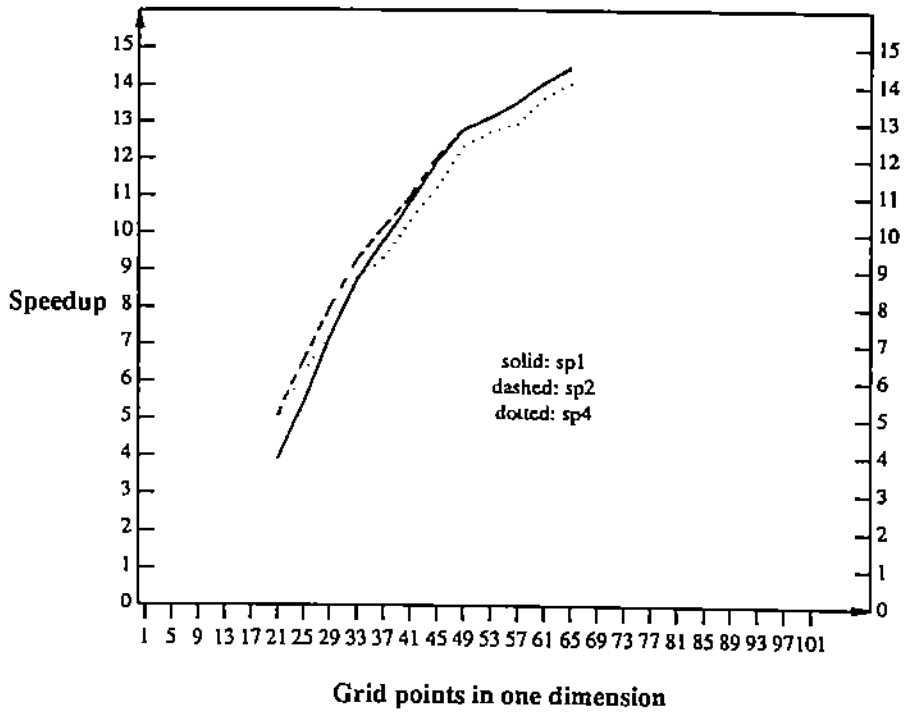

Figure 9. Effect of block wrapping size on the speed up achieved on the iPSC/2 using 16 processors. Curves for wrapping block sizes of 1, 2 and 4 are shown.

VI. Detailed Performance Analysis

We present some excerpts of the details of the performance for a few problems. The data are for 16 processors and are presented in two forms. One is the **execution phases** plot which provides a time history of the computation on each processor. The execution phases are:

Subdomain:	Factor the interior, leaf node submatrix
Compute B2:	Apply factors to the remaining matrix elements of these equations.
Compute D:	Compute the matrix D
Local, global:	Factor D

For very large problems the first phase dominates the computation. This plot does not exhibit the communication costs, so the second form is the **computation load** plot which separates the wait times out and places them at the end of the time history. We also show the density of the communication between processors for a few cases.

Note that the Intel machines have a global clock which is not always uniform across the processors. For example, in Figure 19 it appears that the starting times of the processors different by as much as 50 or 75 milliseconds while they actually start much closer together. This timing inaccuracy creates instances in Figure 25, for example, where messages are read by processor 1 before they are written by other processors (e.g., processors 0 and 3).

Figure 10 shows the real timing behavior at each stage for each processor in solving a 21 × 21 grid problem with wrapping block size = 1 on the iPSC/860. In Figure 11, we subtract from the whole time the idle time spent in each block read while waiting for a message from another processor. Then the total idle time is put at the right end slot for each processor. From these two figures, one can see how much idle time (due to the waiting for communication) each processor spends in each stage and in the whole process. One also sees how the computation is distributed among processors as well as the load balance. Figure 12 shows the effect of changing the wrapping block size to 5 for the same problem on the iPSC/860 and we see that the real performance is substantially improved although the load balance is severely destroyed. For comparison, we show in Figure 13 the same problem running on the NCUBE/2 and observe that the communication takes a much smaller portion than for the iPSC/860. Figures 14 through 18 show the behaviors for a 57 × 57 grid problem with various wrapping block sizes on different machines. We again see that the NCUBE/2 and iPSC/2 have very similar performance while the iPSC/860 is much different. We also observe that with the growth of the problem size the first stage starts to dominate the whole process, which is totally parallelized, while the second and third stages have less effect. These are two important characteristics of the new organization of Gauss elimination.

Figures 19 through 21 show the performance of the iPSC/860 for an 81 × 81 grid problem with different wrapping block sizes. We observe that with the growth of the problem size, then changing the wrapping block size has less effect on the load balance and speed up. Finally, we include Figures 22 through 25 to show the communication paths corresponding to the message receiving points in processor 1 during the fourth stage of factorizing the **Schur complement** D. From Figures 22 and 23 we observe a substantial reduction in the communication for small problems, and therefore a big improvement in the performance. However, from Figures 24 and 25 we see that even though the communication is much reduced for large problems, it is still too frequent, and therefore the performance is not improved very much.

VII. Conclusions

The performance of a parallel algorithm is often architecture dependent. In this report, we have studied the performance of the parallel sparse solver of the Parallel Ellpack System on several up-to-date hypercube computers. It is found that the program which runs in almost perfect speed up on the NCUBE/2 and iPSC/2 shows disappointing speed up on the new iPSC/860. This is particularly true when the problem size is small as there is actually no performance gain by using the parallel processor. The primary reason lies in the much worse computation to communication ratio of the iPSC/860.

By detailed timing of the execution of the program and the explorations of the machine parameters, we located the bottle-necks of the algorithm for the iPSC/860. A block-wrapping assignment and message merging method is used to raise the computation granularity and reduce communication frequencies. Experiments show that this method is very effective in improving the performance when the problem size is small. This method also achieves some performance gain on the iPSC/2 hypercube, although much less effective as on the iPSC/860. The resultant performance is quite satisfactory and looks better than previously reported results for similar problems.

Despite the lower efficiency of the iPSC/860 machine, the machine is much more powerful than other machines when the speed up is reasonably good.

References

1. Ashcraft, C. et al., (1990), "A Fan-in Algorithm for Distributed Sparse Numerical Factorization", *SIAM J. Sci. Stat. Comput.*, **11**, 1990, pp. 593-599.
2. Bokhari, S., (1990), "Communication Overhead on the Intel iPSC/860 Hypercube", ICASE Interim Report 10, May 1990.
3. Bomans, L. and D. Roose, (1989), "Communication Benchmarks for the iPSC/2", *Hypercube and Distributed Computers*, F.Andre and J.P.Verjus (Editors), North Holland 1989, pp. 93-103.
4. Dunigan, T.H., (1990), "Performance of the Intel iPSC/860 Hypercube", Tech. Rept. ORNL/TM-11491, Oak Ridge National Laboratory, June 1990.
5. Geist, G.A. and M.T. Heath, (1986), "Matrix Factorization on a Hypercube Multiprocess", in *Hypercube Multiprocessors 1986*, (M.T. Heath, editor), SIAM Publications, pp. 161-180.
6. George, A., J. Liu, and E. Ng, (1987), "Communication Reduction in Parallel Sparse Cholesky Factorization on a Hypercube", in *Hypercube Multiprocessors* (1987), (M. Heath, Ed.), SIAM Publications, Philadelphia, PA, pp. 576-586.
7. Houstis, E. and J.R. Rice, (1989), "Parallel Ellpack: An Expert System for Parallel Processing of Partial Differential Equations", *Math. Comp. Simulation*, **31**, 1989, pp. 497-508.
8. Mu, M. and J.R. Rice, (1989), "LU Factorization and Elimination for Sparse Matrices on Hypercubes", in *4th Conference on Hypercube Concurrent Computers and Applications*, (Monterey, CA, March, 1989), Golden Gate Enterprises, Los Altos, CA (1990) pp. 681-684.
9. Mu, M. and J.R. Rice, (1990a), "The Structure of Parallel Sparse Matrix Algorithms for Solving Elliptic Partial Differential Equations on Hypercubes", Tech. Rept. CSD-TR-976 (CER-90-19), Department of Computer Sciences, Purdue University, 1990.
10. Mu, M. and J.R. Rice,, (1990b), "A New Organization of Sparse Gauss Elimination for Solving PDEs", CSD-TR-991, CER-90-22, Department of Computer Sciences, Purdue University, 1990.
11. Mu, M. and J.R. Rice, (1991), "A Grid Based Subtree-Subcube Assignment Strategy for Solving PDEs on Hypercubes", *SIAM J. Sci. Stat. Comp.*, to appear.
12. Seidel, S., M.-H. Lee, and S. Fotedar, "Concurrent Bidirectional Communication on the Intel iPSC/860 and iPSC/2", Tech. Rept. CS-TR 90-06, Michigan Technology University, Nov. 1990.
13. Zmijewski, E. (1989), "Limiting Communication in Parallel Sparse Cholesky Factorization", TRCS89-18, Department of Computer Sciences, University of California, Santa Barbara, CA.

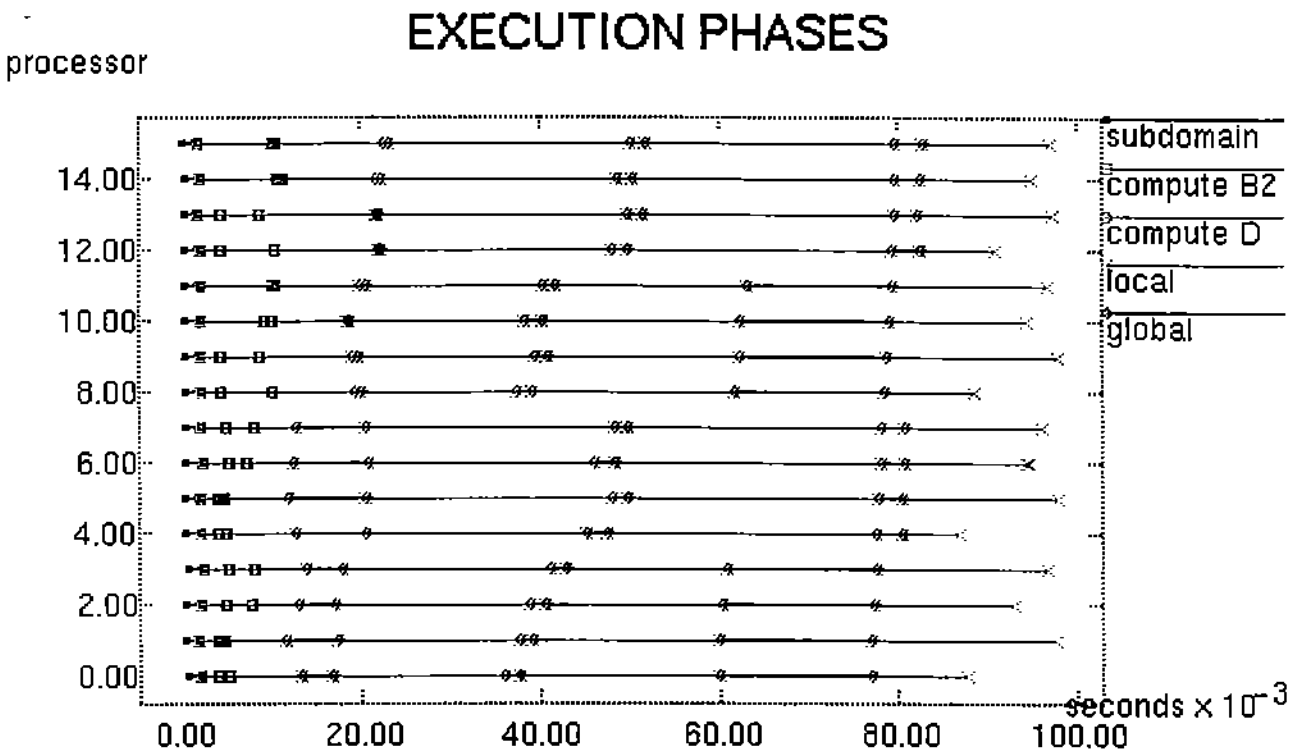

Figure 10. Performance for a 21 × 21 grid problem with the wrapping block size = 1 on the Intel iPSC/860.

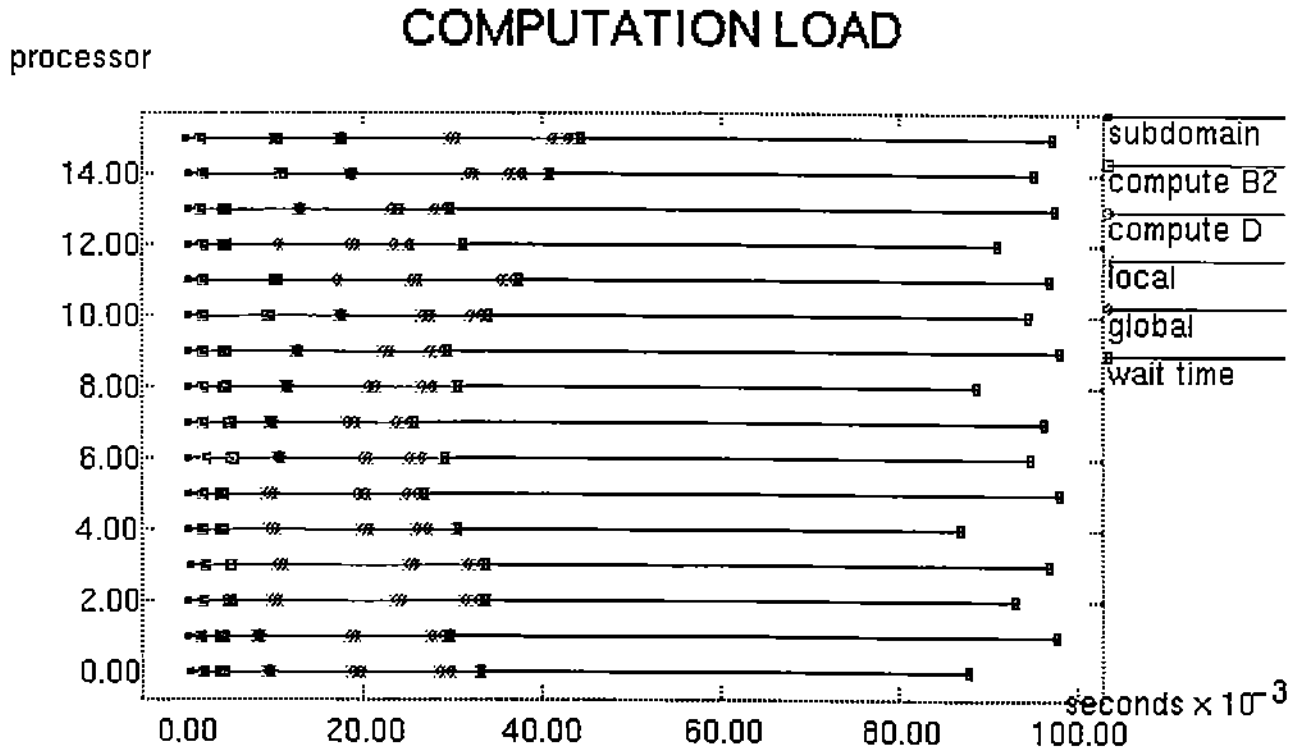

Figure 11. Performance for a 21 × 21 grid problem with the wrapping block size = 1 on the Intel iPSC/860.

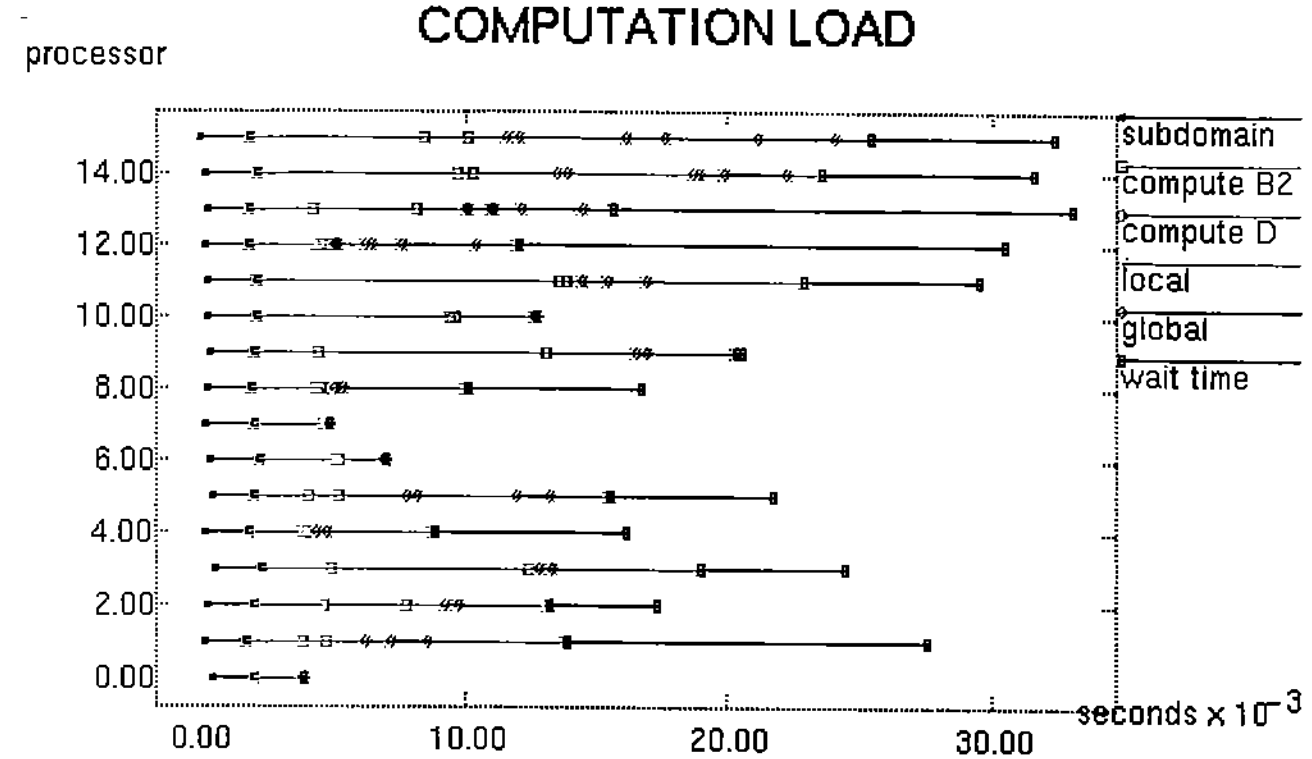

Figure 12. Performance for a 21 × 21 grid problem with the wrapping block size = 5 on the Intel iPSC/860.

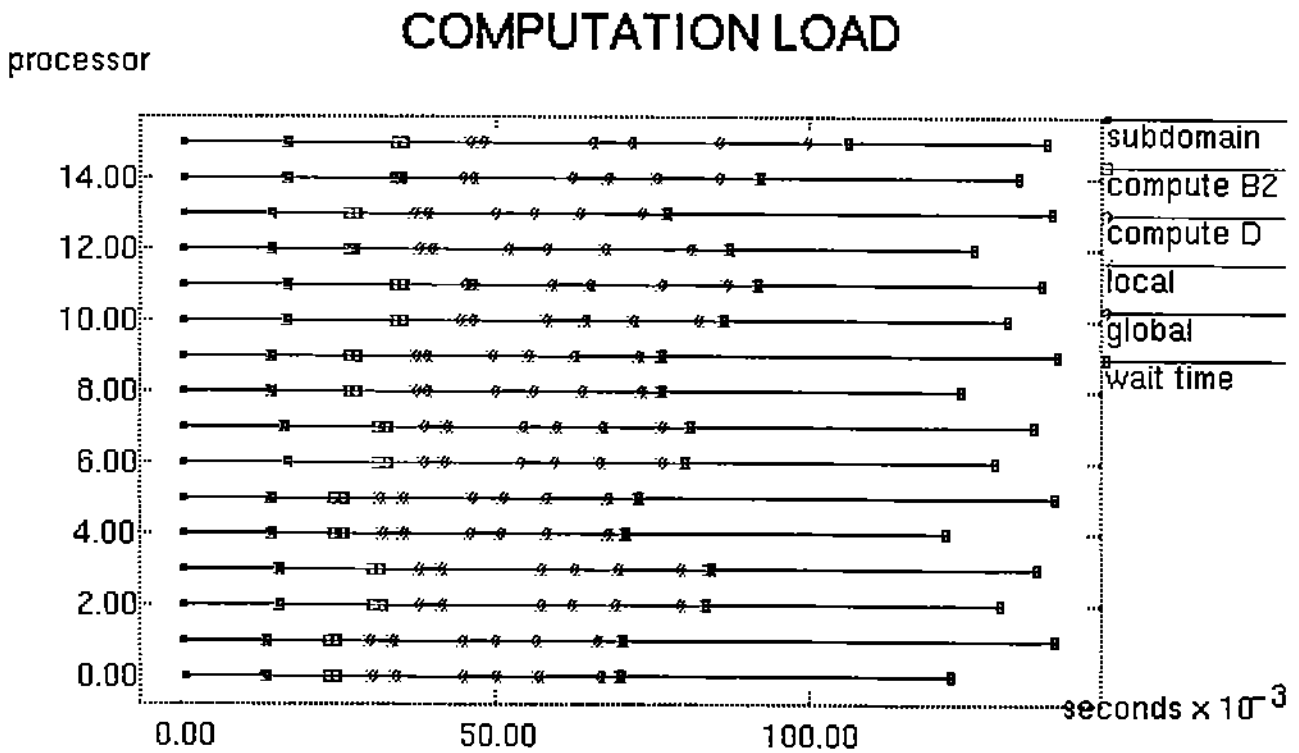

Figure 13. Performance for a 21 × 21 grid problem with the wrapping block size = 1 on the NCUBE/2.

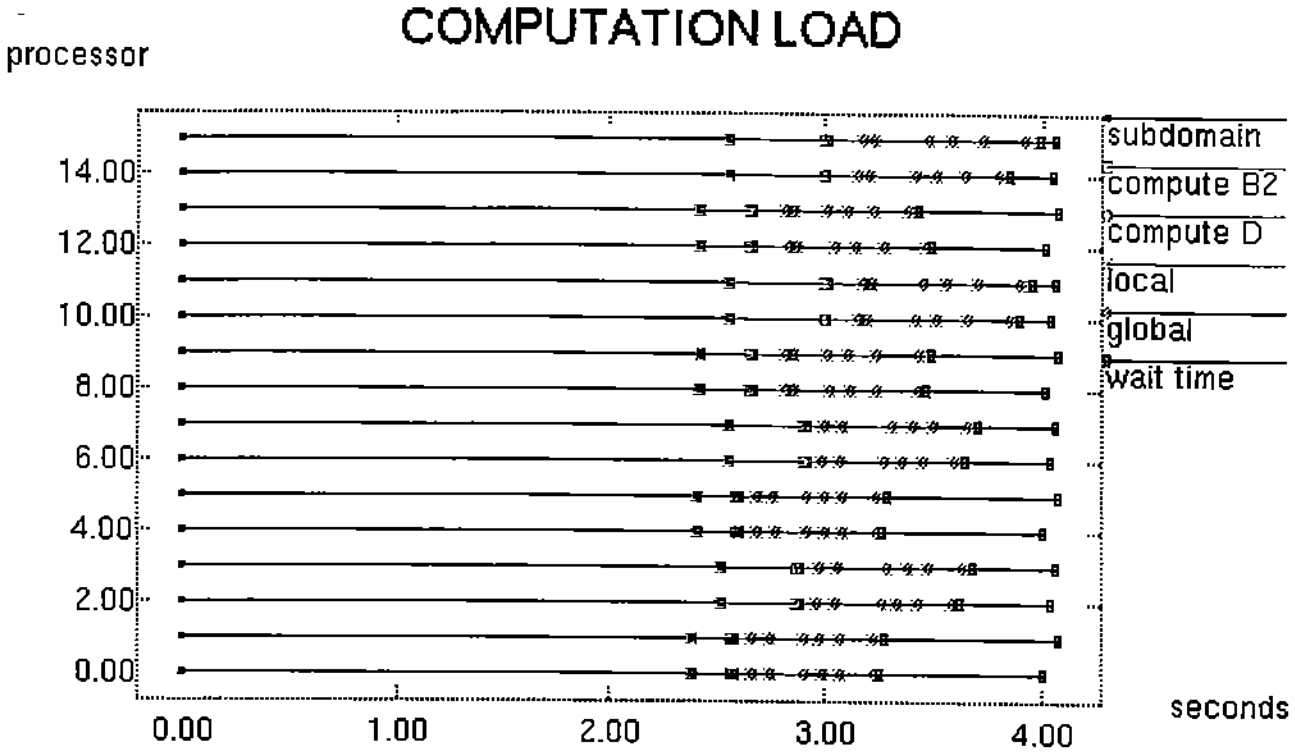

Figure 14. Performance for a 57 × 57 grid problem with the wrapping block size = 1 on the NCUBE/2.

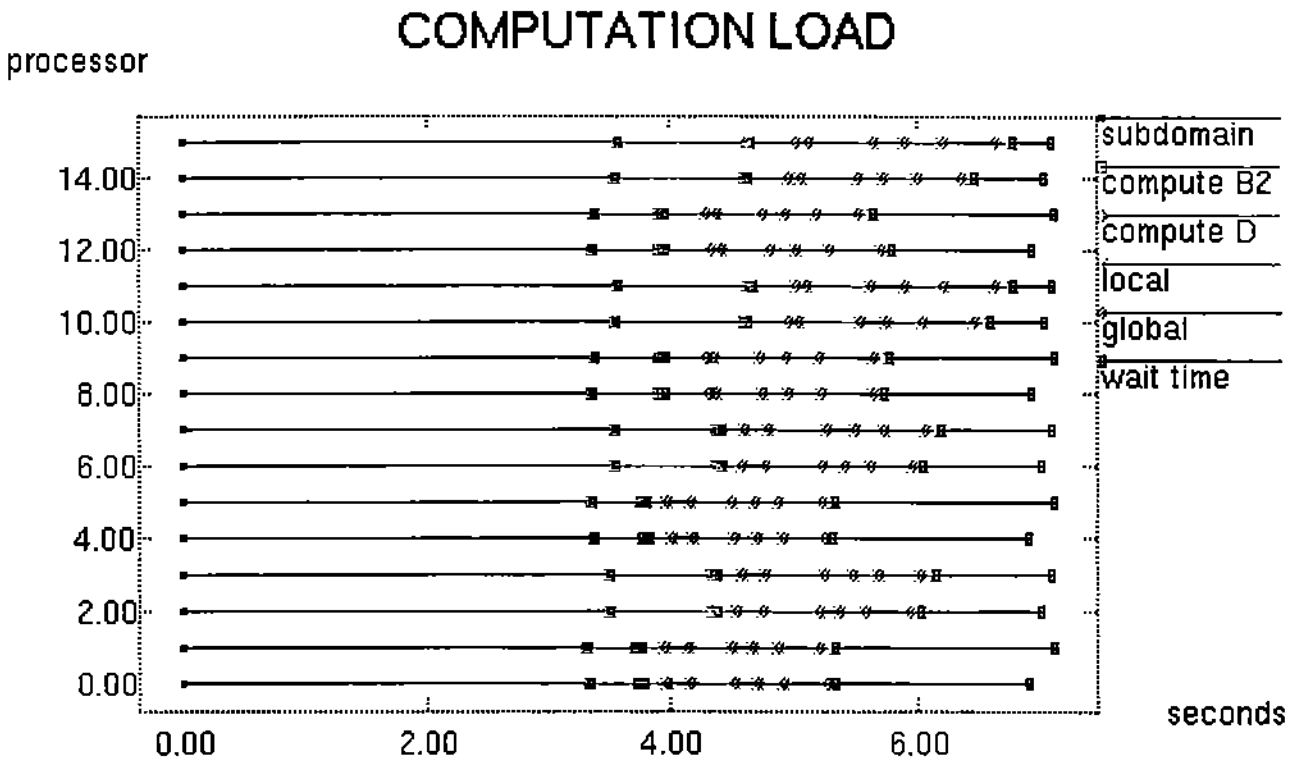

Figure 15. Performance for a 57 × 57 grid problem with the wrapping block size = 1 on the Intel iPSC/2.

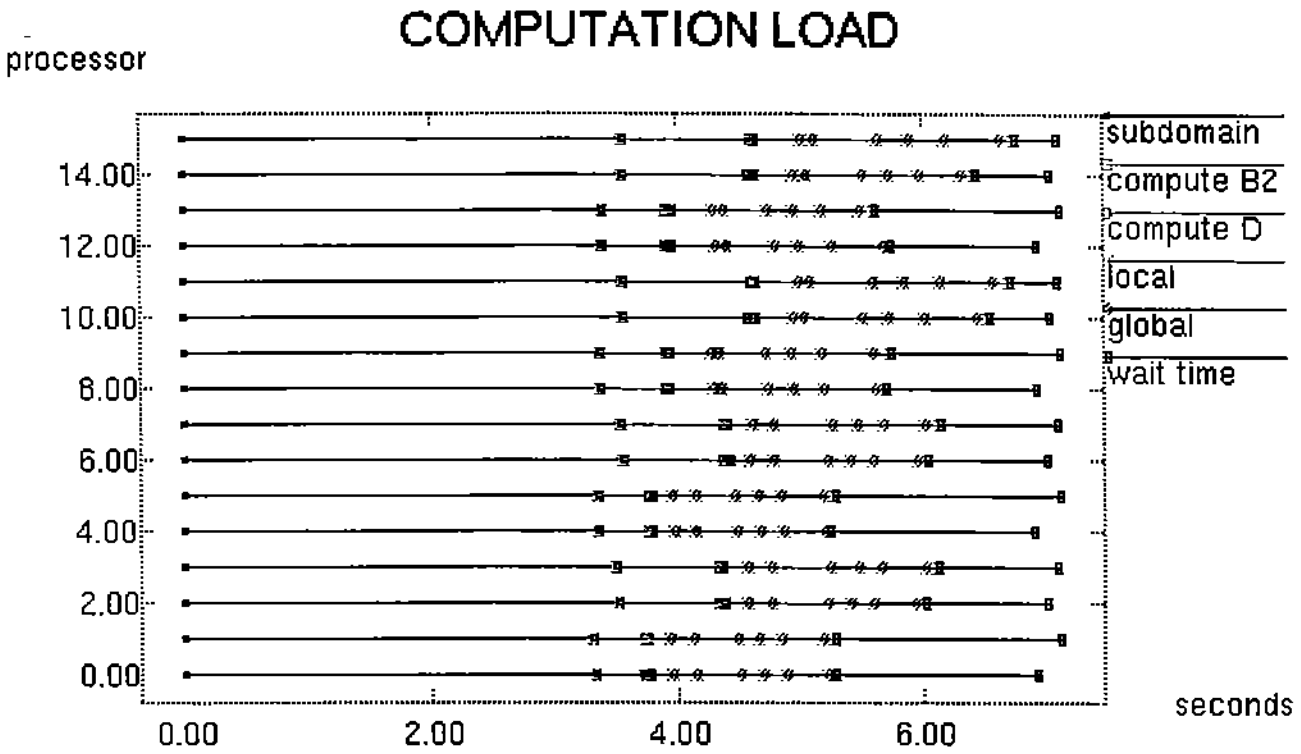

Figure 16. Performance for a 57 × 57 grid problem with the wrapping block size = 2 on the Intel iPSC/2.

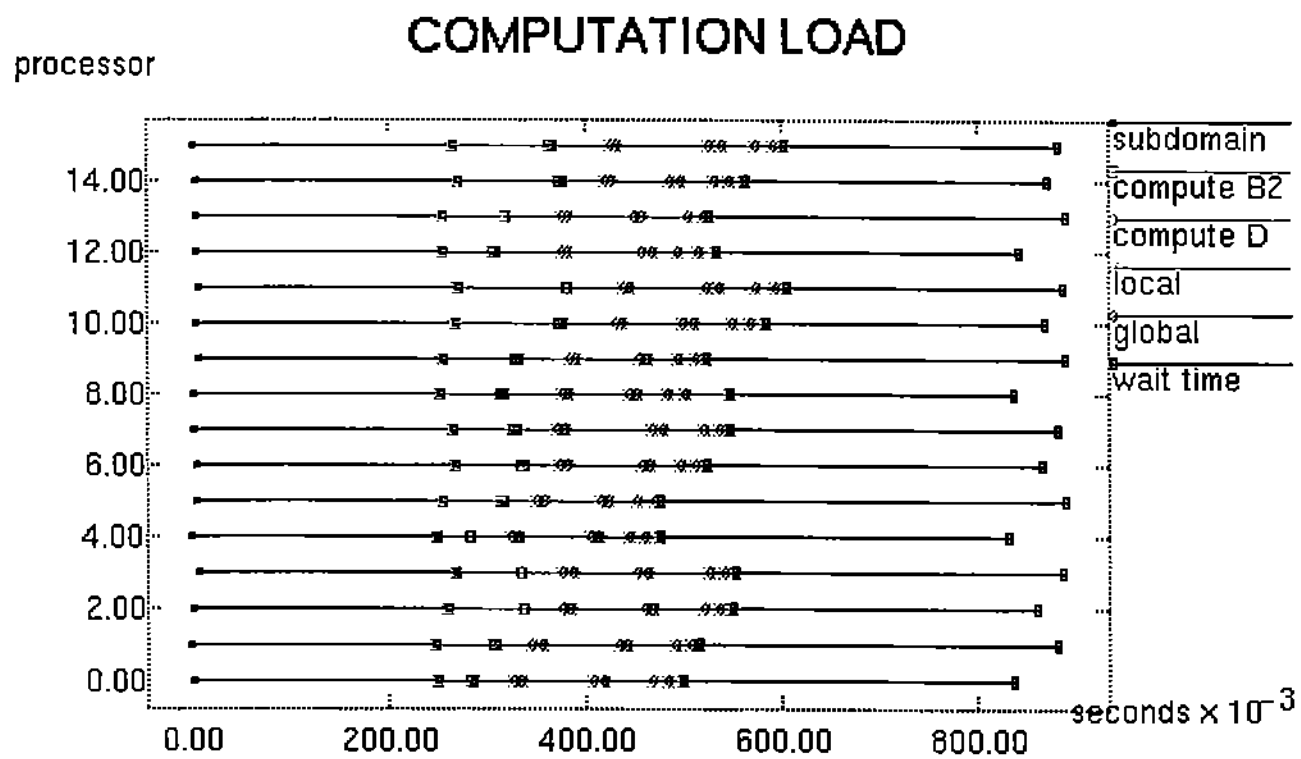

Figure 17. Performance for a 57 × 57 grid problem with the wrapping block size = 1 on the Intel iPSC/860.

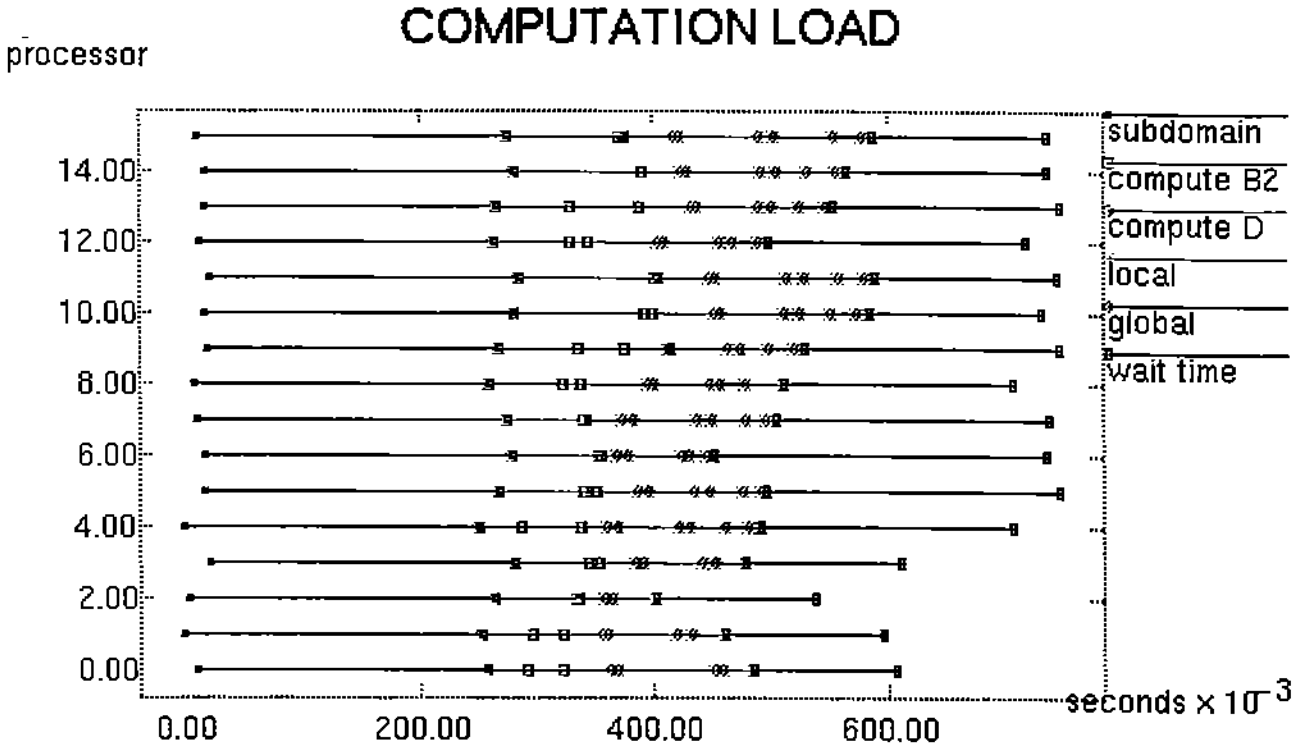

Figure 18. Performance for a 57 × 57 grid problem with the wrapping block size = 5 on the Intel iPSC/860.

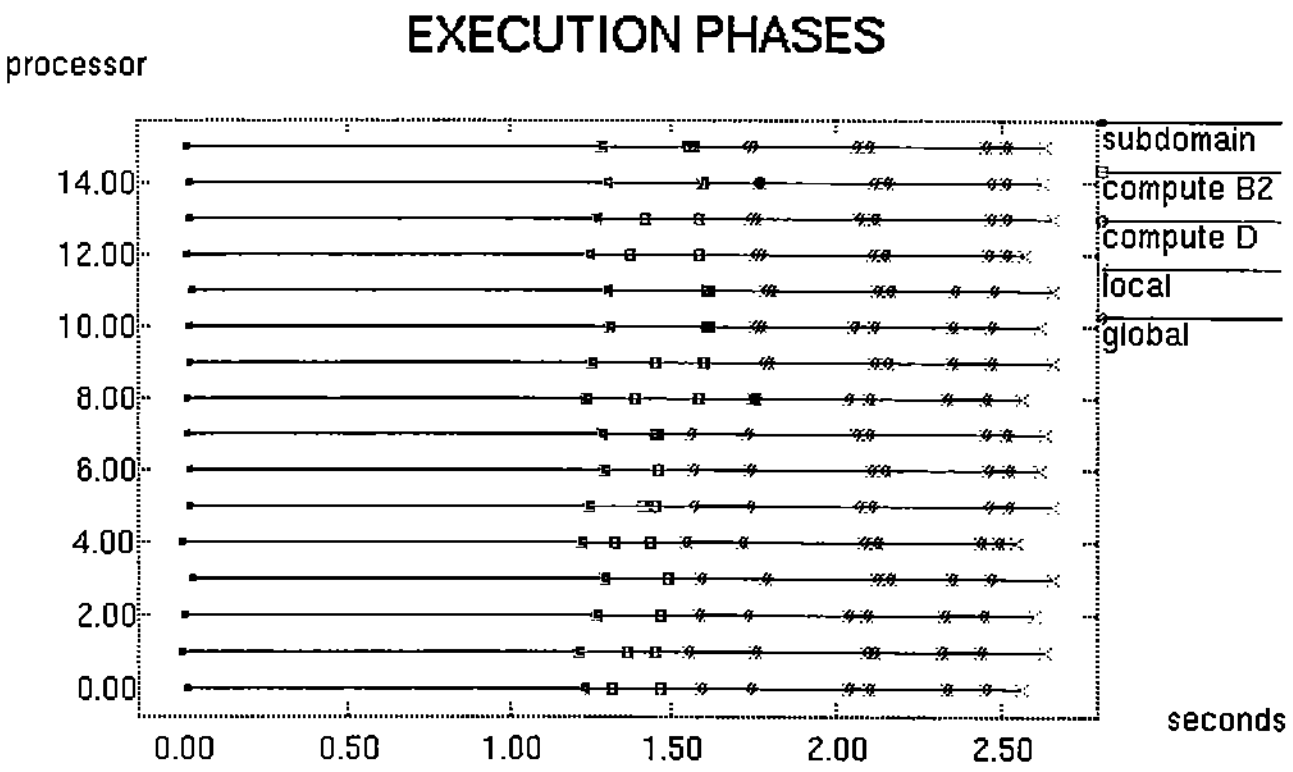

Figure 19. Performance for a 81 × 81 grid problem with the wrapping block size = 1 on the Intel iPSC/860.

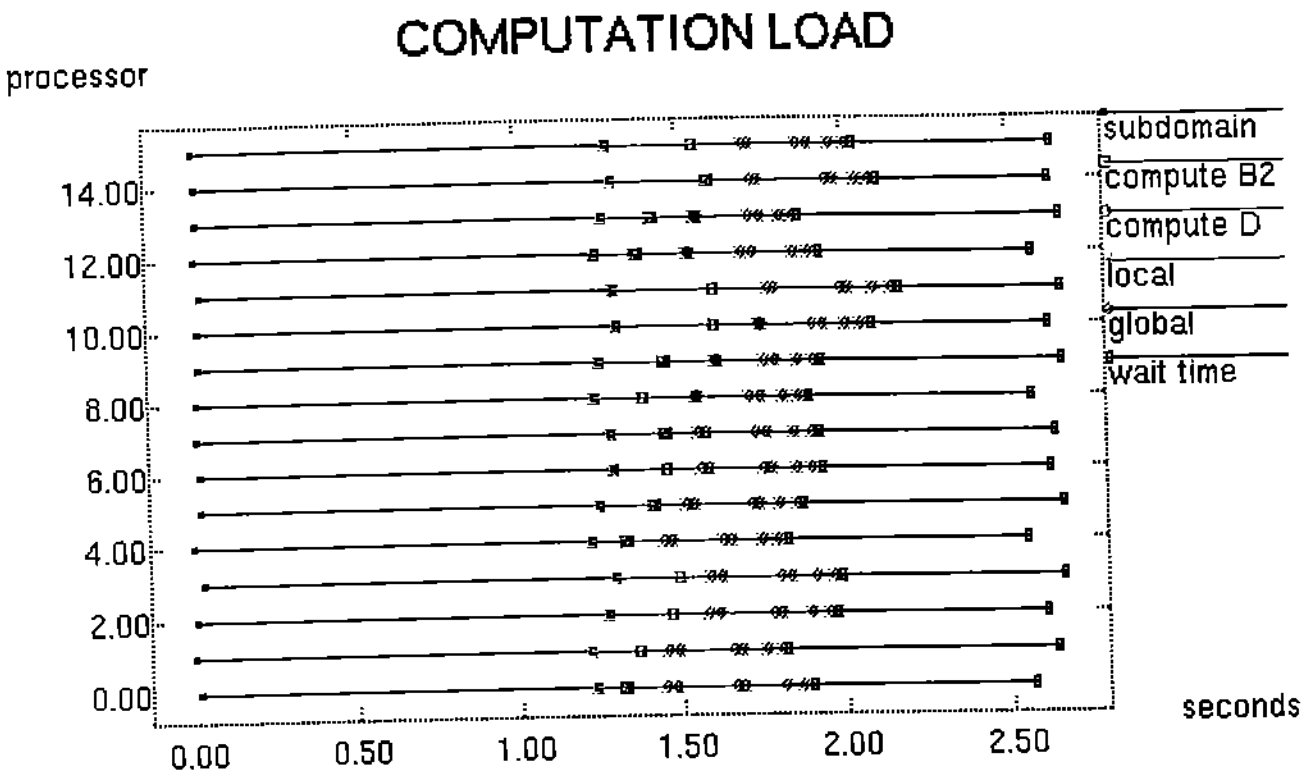

Figure 20. Performance for a 81×81 grid problem with the wrapping block size = 1 on the Intel iPSC/860.

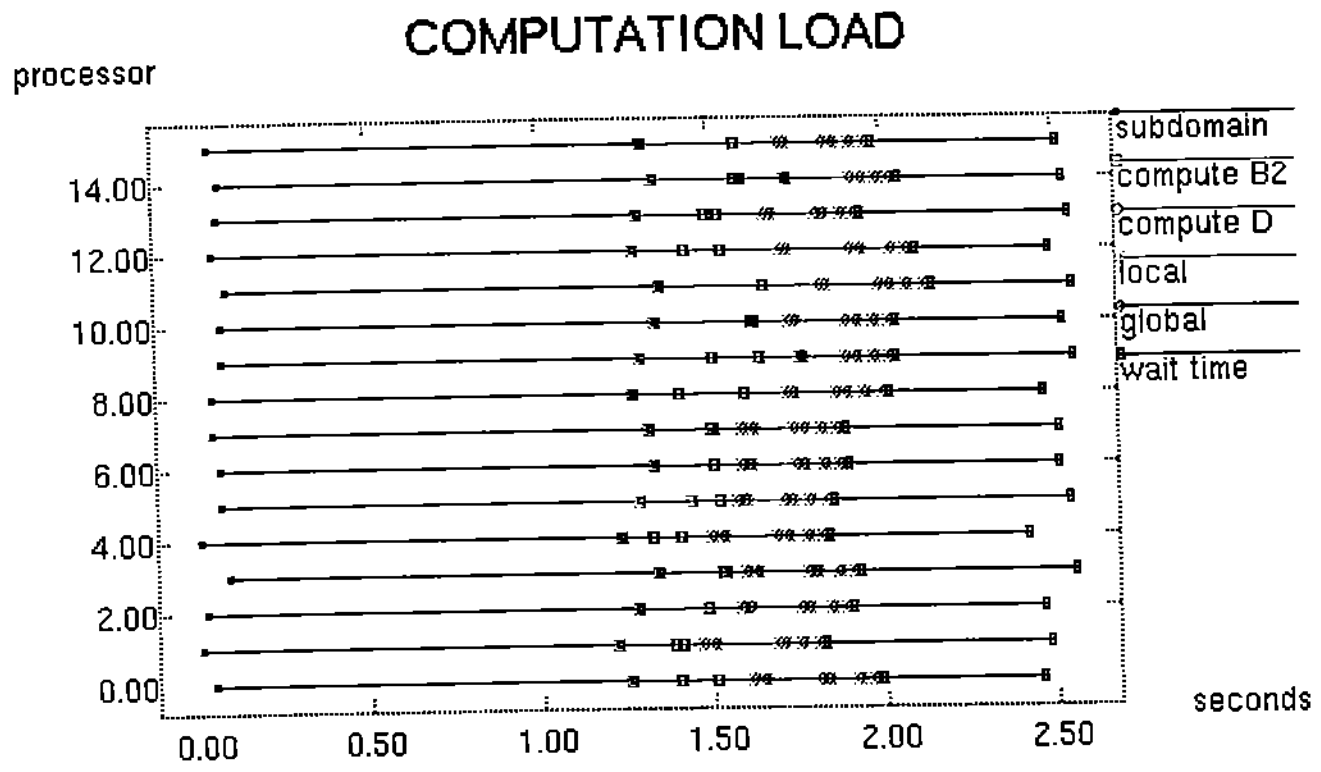

Figure 21. Performance for a 81×81 grid problem with the wrapping block size = 5 on the Intel iPSC/860.

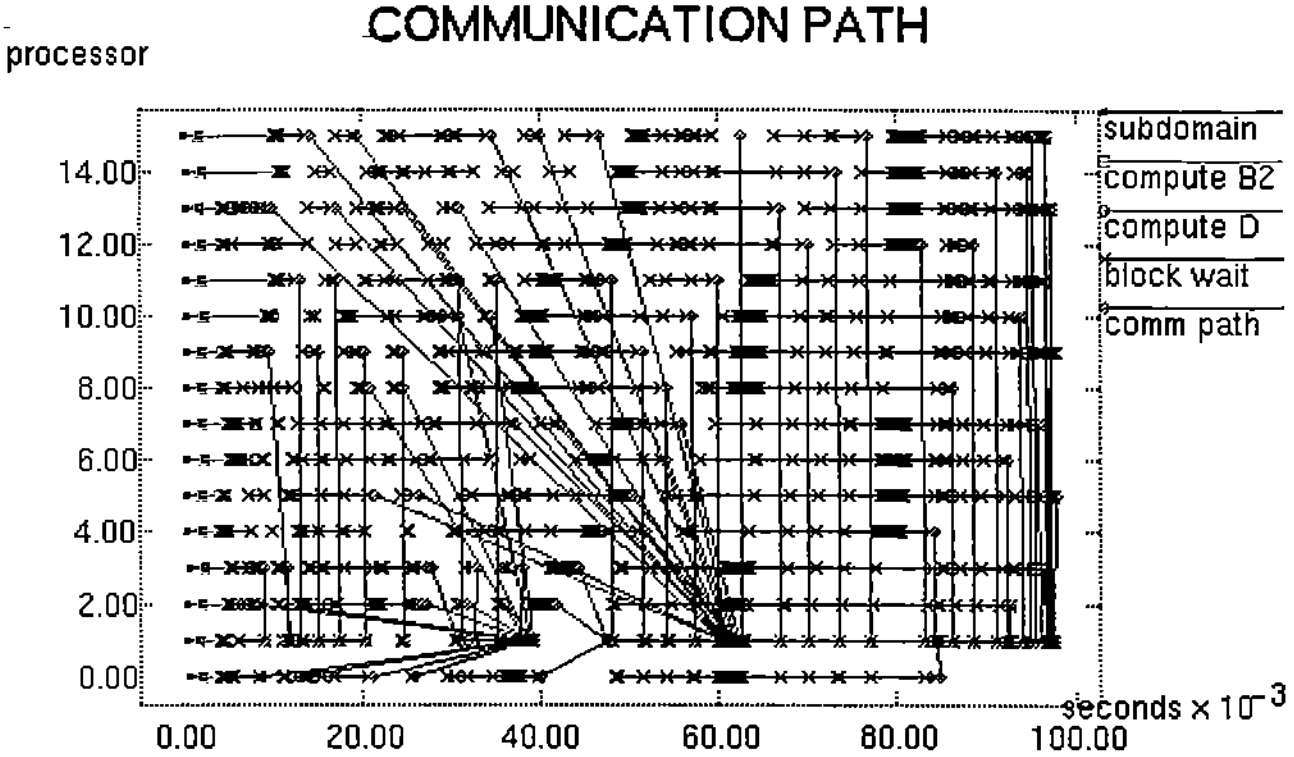

Figure 22. Communication paths to processor 1 for a 21 × 21 grid problem with the wrapping block size = 1 on the Intel iPSC/860.

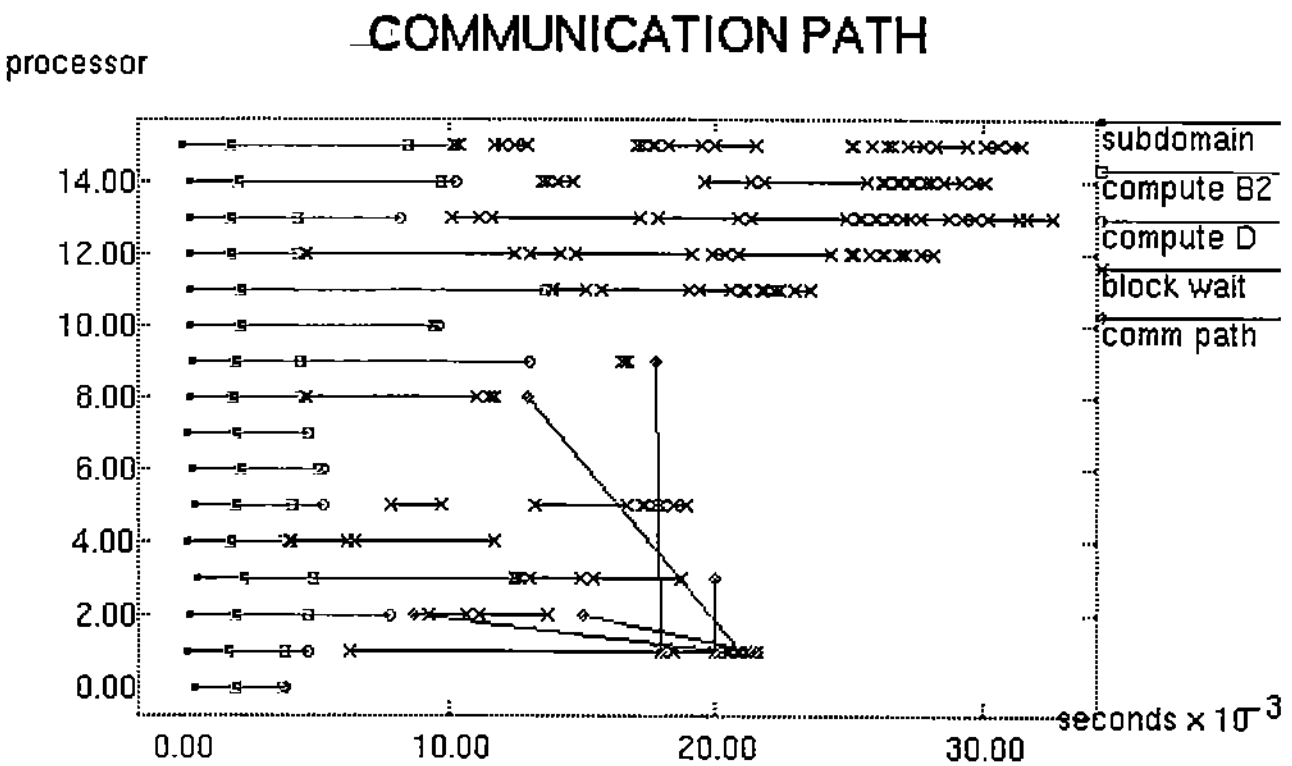

Figure 23. Communication paths to processor 1 for a 21 × 21 grid problem with the wrapping block size = 5 on the Intel iPSC/860.

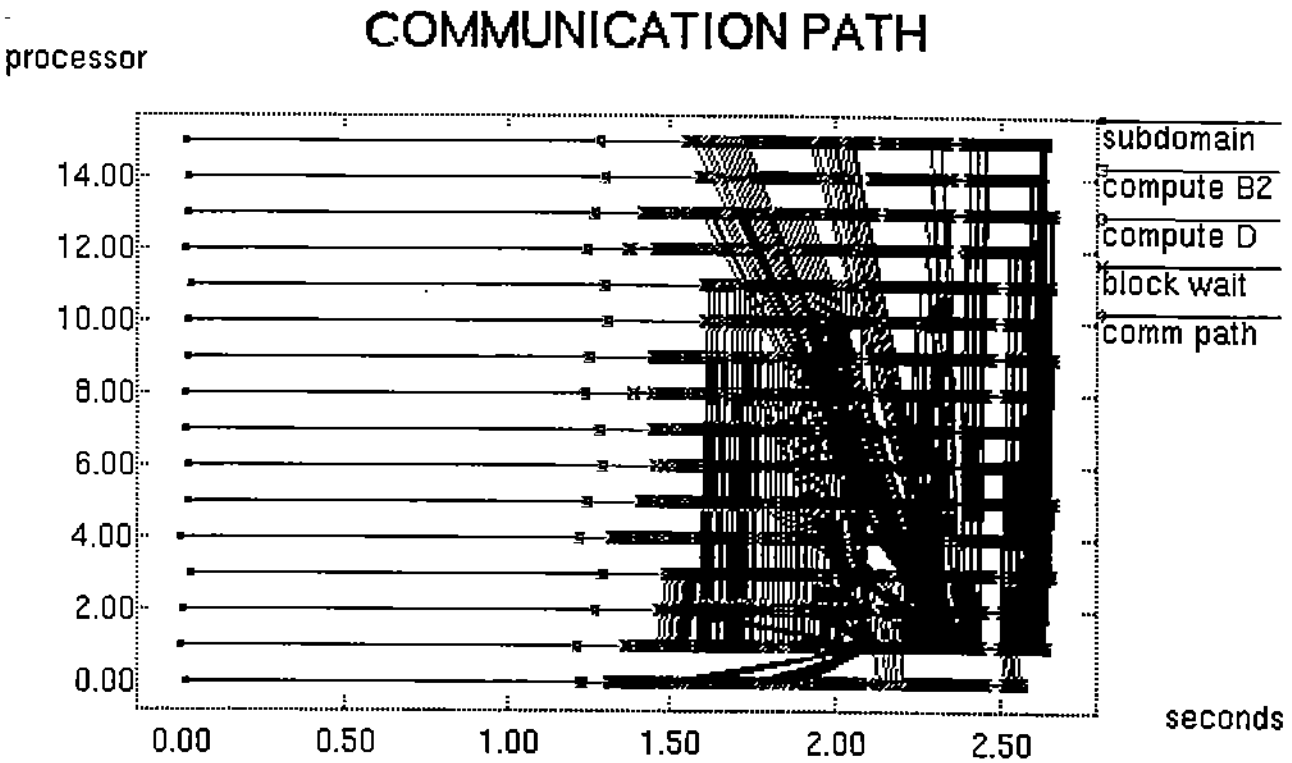

Figure 24. Communication paths to processor 1 for a 81 × 81 grid problem with the wrapping block size = 1 on the Intel iPSC/860.

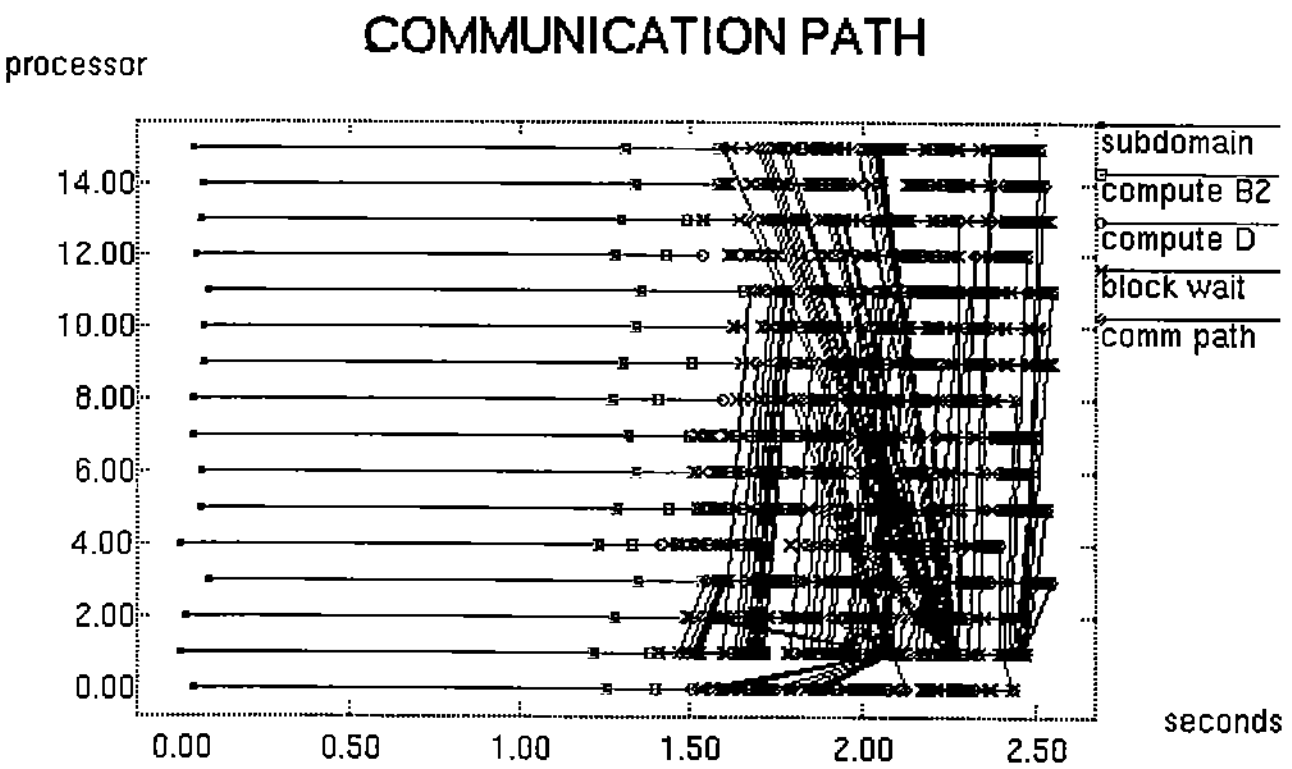

Figure 25. Communication paths to processor 1 for a 81 × 81 grid problem with the wrapping block size = 5 on the Intel iPSC/860.