1992

A PDE Sparse Solver Benchmark for Massively Parallel Distributed Memory Multiprocessors

Mo Mu

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

Report Number:
92-018

A PDE SPARSE SOLVER BENCHMARK FOR MASSIVELY PARALLEL DISTRIBUTED MEMORY MULTIPROCESSORS

Mo Mu
John R. Rice

CSD-TR 92-018
April 1992

A PDE Sparse Solver Benchmark for Massively Parallel Distributed Memory Multiprocessors

Mo Mu* and John R. Rice†
Computer Science Department
Purdue University
Technical Report CSD-TR-92-018
CAPO Report CER-92-08
April 1992

Abstract

This paper investigates the performance of massively parallel, distributed memory multiprocessors for solving elliptic partial differential equations. Our benchmark is a parallel sparse solver of Gauss elimination based on the nested dissection technique. The three machines compared are the nCUBE 2, the Intel iPSC/860, and the Touchstone DELTA. It is shown that these supercomputers are still very far away from being well utilized in this typical scientific application area, even though substantial speed can be achieved as the computing power increases.

1 Introduction

Parallel computers have been developing so rapidly in recent years that the most powerful supercomputers, such as the Intel Touchstone DELTA system, have been able to provide in excess of tens of GFLOPS peak rate and tens of GBYTES main memory. A 13.9 GFLOPS peak LINPACK performance is reported on the DELTA for solving a $25,000 \times 25,000$ dense linear system of equations using all 512 nodes [1], also reported is the LINPACK benchmark for over a hundred computers. Dunigan [3] reports the communication performance of the DELTA, the iPSC/860 and the nCUBE 2 using several simple and commonly used models. We [10] also report the speedup and efficiency performance of a dense matrix solver on the nCUBE 1. This paper investigates the potential performance of these emerging supercomputers in a more realistic, and more complicated, application from scientific computing and examines the impact of the machine evolution on real applications.

Our experiments involve solving elliptic boundary value partial differential equations (PDE). The resulting sparse linear systems of equations are solved by Gauss elimination using the nested

*Supported by NSF grant CCR-86-19817.
†Supported in part by AFOSR grant 88-0243 and the Strategic Defense Initiative through ARO contract DAAG03-90-0107.

dissection technique. The parallelism is realized by the domain decomposition which generates block structured matrices. Two important characteristics of this problem are its sparsity and irregularity. The former implies the fine granularity of the organization of computation and communication, and the latter causes the complexity in both algorithmic and communication organizations. It is, therefore, a challenging problem to achieve high performance for this benchmark on the currently available massively parallel, distributed memory multiprocessors. These machines have slow communication rates relative to their computation rates and thus favor coarse-grained problems. Note that this application involves matrices requiring many fewer numerical operations than those huge dense, (or even band), matrix problems where nearly peak performance (in FLOPS) is achieved. In practice, a 100×100 grid is usually fine enough to solve a two-dimensional PDE, which gives a sparse linear system of 10,000 equations and perhaps 0.05% non-zero elements in the linear system. Three-dimensional problems can get a lot larger, having about 1 million equations, but are even more sparse with perhaps 0.002% non-zeros. On the other hand, handling the sparsity requires a big overhead for the data structure manipulation. Note that the interest here is to have high solution speed, instead of high performance in FLOPS. Our experiments are performed on three machines, the nCUBE 2, the Intel iPSC/860, and the Touchstone DELTA.

The paper is organized as follows. Section 2 provides a summary of the hardware specifications of these three machines, followed by a description of the benchmark problem and the algorithm. In Section 3 we illustrate the performance of these machines from various aspects using the PDE sparse solver benchmark. The paper concludes with some observations and future work directions. For readers' convenience, we finally provide all of our timing data in the appendix.

2 Environments and Test Problems

2.1 Hardware Specifications

We list in Table 1 the hardware specifications of three machines used in our experiments. They are all distributed memory, message passing MIMD multiprocessors. The nCUBE 2 and the iPSC/860 used are located at Purdue University, and the DELTA system is at the California Institute of Technology. The DELTA system has a total of 528 compute nodes (a 16×33 mesh), but the largest mesh available to an application is 16×32. The MFLOPS entries in column 6 refers to the peak single precision floating-point operation rate in each individual node since all of our computations are in single precision. In column 8, the iPSC/860 requires twice as much start-up time for a message larger than 100 bytes as for a short message due to its buffer-request/reply protocol. The DELTA communication protocol has some drawbacks because it breaks a long message into 476-byte based segments, and, for messages longer than 6 segments, a buffer-request/reply protocol is also used. In addition, the hop penalty is almost negligible on the DELTA for remote communications. All this information is found in the manufacturer's documentation or [2] and [3].

2.2 PDE Sparse Solver

We use a generic parallel PDE sparse solver for general elliptic boundary value problems using any potential discretization along with domain decomposition and nested dissection techniques.

Table 1: Hardware specifications for the three machines. The MFLOPS entry in column 6 is peak performance for single precision. The startup times in the last column are in microseconds, the parenthesized value for the iPSC/860 is for messages of over 100 bytes.

Machine	Architecture	Node processor	Number of processors	Memory per node	Computation rate	Communication rate	Start-up time
nCUBE 2	hypercube	Custom 64 bit	64	4 MB (16 nodes) 1 MB (48 nodes)	7.5 MIPS, 2.5 MFLOPS	3.3 MB/S	154 μS
iPSC/860	hypercube	i860	16	16 MB	33 MIPS, 80 MFLOPS	2.8 MB/S	75 μS (136 μS)
DELTA	mesh	i860	512	16 MB	33 MIPS, 80 MFLOPS	22 MB/S	72 μS

The linear system of equations can be nonsymmetric. However, for the purpose of a benchmark, we choose a standard model problem and the *5-point-star* discretization so that others can easily compare our data with other machines and with software that is not so general. Without loss of generality, the algorithm is also described in the model problem context. The model problem is to solve a Poisson equation with Dirichlet condition on a rectangular domain. Suppose that $p = m^2$ processors are used to solve the problem. The domain is decomposed by an $m \times m$ square mesh. A one-to-one subdomain to processor mapping is established such that geometrically adjacent subdomains are mapped to physically direct neighbor processors. This is realized by a tensor product *gray-code* mapping for a hypercube and by a trivial mapping for a mesh. An $n \times n$ tensor product discretization grid is then imposed on the domain consistent with the domain decomposition so that the decomposition interfaces lie on grid lines. We use *5-point-start* (standard finite differences) to discretize the PDE problem. A linear system of equations is then generated with an *incomplete nested dissection* ordering. The ordering is based on the domain decomposition that the standard nested dissection is first applied to each subdomain, and then the interface grid points are ordered separator by separator nested in alternate directions. The corresponding matrix is thus of a nested block diagonal, bordered (or arrow oriented) structure. The linear system, in turn, is solved by a *parallel implicit block Gauss elimination* algorithm [6]. No pivoting is used. The equations are distributed among processors using a *grid-based subtree-subcube* (or submesh) assignment [9]. It first assigns grid points in each subdomain to the corresponding processor, and then assigns grid points in each separator, segment by segment (in the mesh decomposition) to the processors of the subcube (or submesh) on the same line where the segment lies. The parallel algorithm is sketched with the following major steps (for details of this algorithm, see [5] [6] [8]).

Sparse Gauss Factorization Algorithm [6]

Step 1. Each processor, $i = 1, 2, \ldots, p$, factors its local subdomain matrix

$$A_i = L_i U_i; \tag{2.1}$$

Step 2. Each processor computes the necessary data in the matrix

$$E_i = A_i^{-1} B_i \tag{2.2}$$

and sends them to processors that will need these data in Step 3. The matrix B_i, local to the processor, is in the column border of the global matrix, which represents the connections of the i-th subdomain to its interfaces;

Step 3. Each processor receives the data sent from other related processors in Step 2 and participates in computing the local part of the Schur complement matrix

$$S = D - \sum_{i=1}^{p} C_i E_i, \tag{2.3}$$

where D is the interface matrix and C_i represents the connection of interfaces to subdomain i;

Step 4. Each processor participates in factoring the Schur complement

$$S = L_s U_s \tag{2.4}$$

using the block elimination tree from the nested dissection. This step consists of a series of local and global eliminations for the tree nodes (or equivalently, the separators).

All these steps are sparse computations. Here we only list the factorization part, and a block back solve algorithm can be formulated similarly. This paper only considers the factorization because so far there is no truly efficient, parallel, distributed triangular sparse solver available. There is no separate symbolic factorization employed in this algorithm, this process is mixed with the numerical factorization and managed by a dynamic data structure. For the sequential benchmark, we use the *zero-tracking* code of the Yale Sparse Matrix Package [4] with the standard nested dissection.

The communications involved in this problem can be characterized as follows. First, messages are usually not very long, and therefore, the peak communication bandwidth is far from being reached on each of these machines; this means that the start-up overhead dominates the communication cost. Second, the communication patterns are rather random and messy due to the dynamic data structure used to track fill-ins during the elimination. A typical communication here is multicasting which sends a message to a list of processors. The destination list is usually short, but

rather scattered, i.e., the processors involved are not necessarily direct neighbors. The operating system support for multicast is not efficient on these machines (this is an implementation, not intrinsic, problem) and thus the overall performance of the algorithm is substantially impaired by this type of communication. Overall, we believe that this benchmark is a very good and typical example to examine the potential performance of these machines in real applications. It has a large degree of inherent parallelism but it also has complexities which make it far from embarrassingly parallel.

3 The PDE Sparse Solver Benchmark Results

The first measurement of interest for a parallel computation is, of course, the speedup. It is defined by the time of running the sequential benchmark code on one node processor of the machine divided by the wall-clock time of running the parallel code using multiple processors to solve the same problem. Figure 1 shows the speedups obtained using 4, 16, and 64 processors (corresponding to 2×2, 4×4 and 8×8 domain decomposition). No data is given for 64 processors on the iPSC/860 as our machine only has 16 processors. We see that, for a given number of processors, the limiting speedup for the nCUBE 2 is much higher, and reached earlier, than for the other two machines. The DELTA's limiting speedup is some higher, and reached earlier than for the iPSC/860. These results primarily reflect how the performance in this application depends on the ratio of communication-to-computation speeds. Also, the limiting speedup on each of these machines can be reached by reasonably increasing the problem size if using no more than 16 processors.

Using 64 or more processors to solve large problems on the nCUBE 2 leads to problems with memory. Because of the limited memory size, the system allocates a small communication buffer of 65,536 bytes. As the number of processors and problem size increase, the size needed for the communication buffer also increases. The default buffer size is only sufficient for problems of size 41×41 grid or smaller when using 64 processors. The nCUBE 2 compiler *ncc* allows users to adjust the communication buffer size. But increasing the communication buffer size reduces the memory available for the program. We can run problems of size up to a 65×65 grid using 64 processors on our nCUBE 2 and can still see the speedup increase.

However, in practice, it is hard to reach the limiting speedups on the DELTA and the iPSC/860 if using more than 64 processors. The memory limitation is no longer severe for these machines, but the grid sizes (perhaps 300 by 300 or larger) required to generate a sufficiently large computation are not commonly needed.

The second interesting measurement is the efficiency defined by the speedup divided by the number of processors used. It reflects the utilization (in time, not memory) of processors. The efficiency is, of course, bounded by 1 (100%). In practice, efficiency is less than 1, even when the communication speed is fast enough that the communication time could be negligible. This loss in efficiency is caused by the following two penalties. First, for achieving parallelism, parallel algorithms usually lose some efficiency that the best sequential algorithm has. Second, parallel algorithms for massively parallel, distributed memory machines have extra overhead for managing information like assignment, scheduling, etc., and for processing the communication messages. It is misleading to compare the parallel time with the time of running the parallel code (instead of

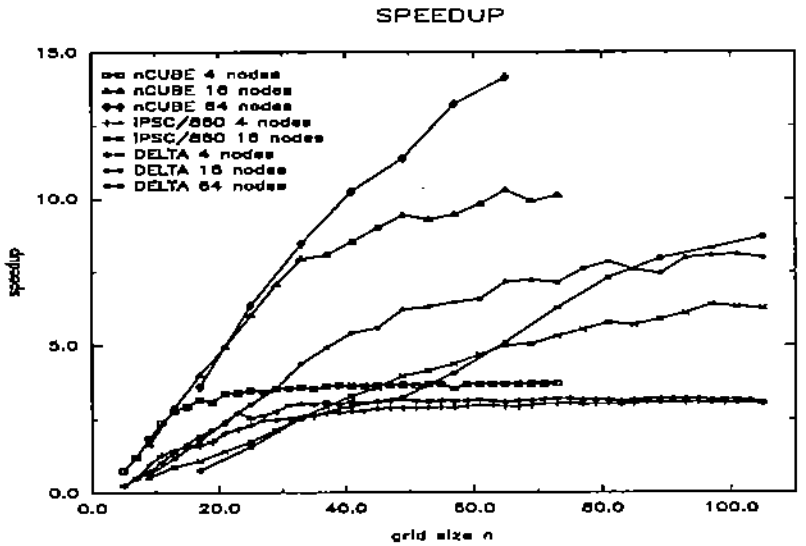

Figure 1: Speedups obtained using 4, 16, and 64 processors (coresponding to 2x2, 4x4, and 8x8 domain decomposition) on the three machines.

the best sequential code) on a single processor. This practice can misrepresent the gain made using multiprocessors to solve a problem. In other words, high efficiency does not simply mean keeping all processors busy but rather means exploiting the potential computing power. Processors should be busy in doing useful things. Figure 2 shows the processor utilization for these machines and in Table 2, these machines are ranked according to their efficiency. For each row, column 4 lists the maximum observed efficiency in tested problems, column 5 gives the corresponding grid size, and column 6 gives our conjecture as to the maximum achievable efficiency ignoring limitations due to memory sizes.

Among our concerns is how much time reduction is gained in solving these problems by using more powerful machines as the number of processors increases. This is reflected in Figure 3 which shows the wall-clock time to solve various problems. Figure 4 gives an enlarged view of the lower left part of Figure 3. For example, we see that using 16 nCUBE 2 processors to solve these problems is almost as fast as using a single high speed i860 processor. Using 64 DELTA processors is only a bit faster than using 16 processors on the same machine (which has over 500 processors). Further, a 64 node DELTA is not faster until the grid size is larger than 81×81. Figure 5 provides another view of the time performance. The time data are divided by the corresponding sequential time on the nCUBE 2 for each grid size. This can also be viewed as a relative measurement of FLOPS; note that it is difficult to analytically count the operations in this type of sparse matrix computation. Let us generically define

Table 2: Efficiency relative to the best sequential algorithm, the conjectured maximum possible (column 6) ignoring memory limitations.

Rank	Machine	Number of Processors	Observed Maximum Efficiency	Corresponding Grid	Conjectured Maximum Efficiency
1	nCUBE 2	4	92.8%	73×73	95%
2	DELTA	4	80.0%	95×95	85%
3	iPSC/860	4	77.0%	87×87	80%
4	nCUBE 2	16	64.3%	65×65	75%
5	DELTA	16	50.7%	101×101	65%
6	iPSC/860	16	39.9%	97×97	50%
7	nCUBE 2	64	22.1%	65×65	30%
8	DELTA	64	13.6%	105×105	25%

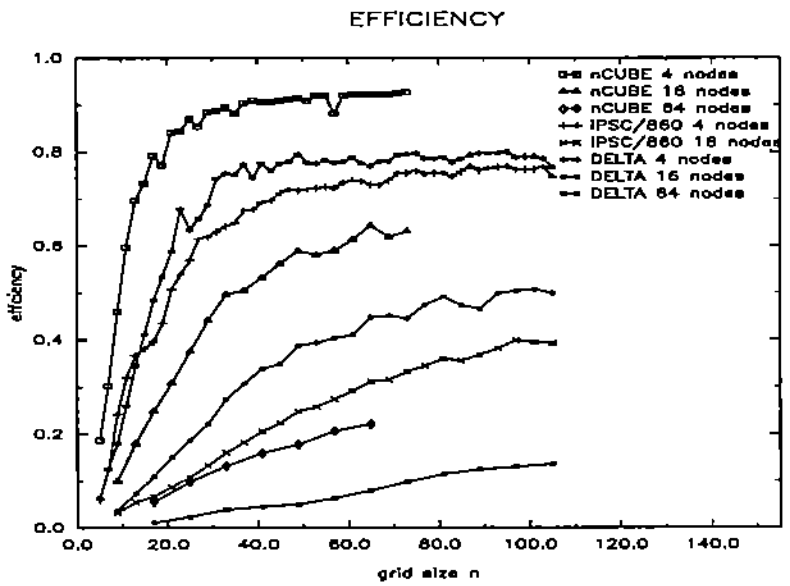

Figure 2: Processor utilization for various numbers of processors and various machines.

$$\text{FLOPS} = \frac{\text{work}}{\text{time}}$$

where *work* is the number of any kind of time consuming operations. Let the subscript r refer to the measurement on a single nCUBE 2 processor, we have

$$\begin{aligned} \frac{\text{FLOPS}}{\text{FLOPS}_r} &= (\text{ work/time })/(\text{ work/time}_r) \\ &= \frac{\text{time}_r}{\text{time}}. \end{aligned}$$

This ratio can also be viewed as a measurement of speedup relative to the nCUBE 2 sequential time. For the nCUBE 2 machine, it is just that.

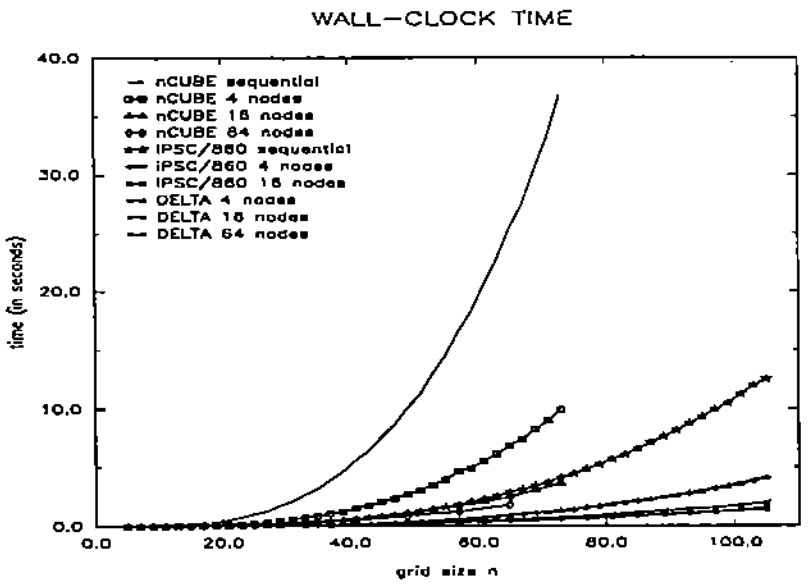

Figure 3: Wall-clock time to solve various problems. See Figure 4 for an enlargement of the lower left part of this figure.

An interesting observation for the iPSC/860 relates to its *send-to-all* facility (obtained by using its communication primitive *csend* with the destination field of value -1). When the number of processors is big, for example, 16 processors in our experiments, there is a substantial improvement by replacing the multicasting with send-to-all in our program, even though it introduces many more messages. This is due to the special implementation of the *sent-to-all* function, so that it requires about the same startup overhead as for a single destination *send* communication. This effect is shown in Figure 6. However, this effect is not observed on the other two machines, even though the DELTA provides the same function call. This illustrates how efficiencies (or the lack thereof) in system implementations can produce counter-intuitive performance results. All our data on the iPSC/860 use the faster *send-to-all.*

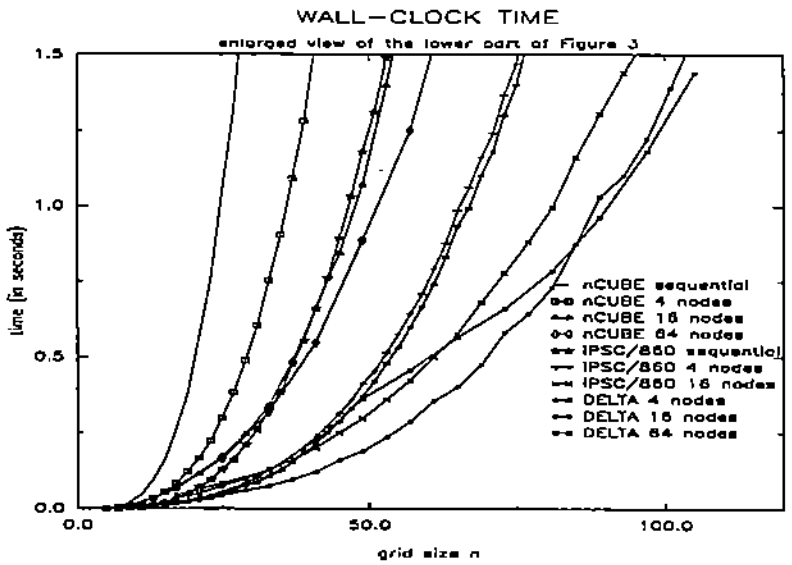

Figure 4: An enlarged view of the lower part of Figure 3 showing the wall-clock time to solve various problems.

Another thing to be noted for the DELTA is that its timing performance is not very stable. For example, we made three runs using 64 processors to solve a 97×97 grid problem at different times and it took 1.18, 1.39 and 1.42 seconds, respectively. The difference is about 17%, which is fairly large. We guess that this variation might be the effect of contention on the communication channels due to other jobs running simultaneously on the mesh.

We use two kinds of pictures to visualize the performance details. One is the *execution phase* which shows the time history of the real execution for each of the major phases of the algorithm. The other is the *computation load* which shows the computation times with the communication times (receives and sends) broken out and moved to the right end of the history. The communication time is obtained by timing before and after each communication subroutine call. The computation time, then, is calculated by subtracting all the communication time in the interesting part from the corresponding real execution time. We use this computation time to indicate the net time required by the computation task in the algorithm. Therefore, the summation of the computation times over all processors should reflect the total computation overhead of the algorithm; this should not depend on different assignments and schedulings. However, one strange phenomenon we have observed is that assignments and schedulings do, nevertheless, substantially affect this counting of the computation time. We still do not know the exact cause of this. It seems to us that when some kinds of communication events, especially sends, happen in one processor, there are some delays in its partner processors. This might be due to the communication protocols and, if so, the effect is hard to measure explicitly. At any rate, these pictures still illustrate the parallel performance even though the computation times might contain other overheads which are not expected in the

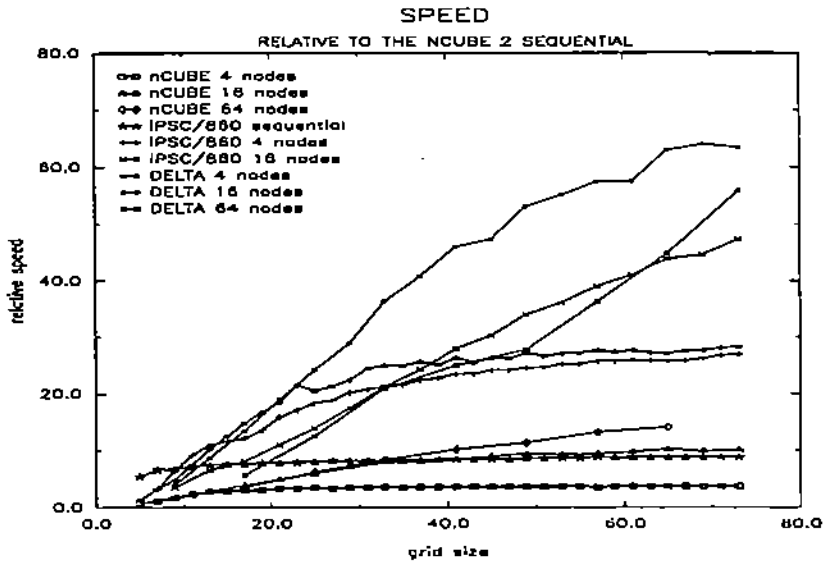

Figure 5: Speed relative to the sequential benchmark running on the nCUBE 2.

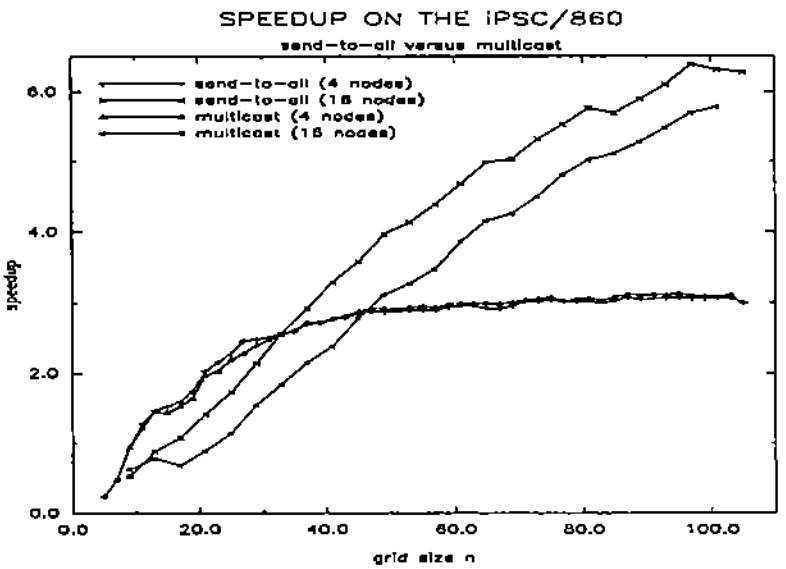

Figure 6: The effect of replacing multicast with send-to-all on the iPSC/860. The effect is quite significant for 16 processors.

sequential computation.

Figures 7.A - 7.F, 8.A - 8.F, and 9.A - 9.F show the nCUBE 2 behavior for solving problems of various sizes using 4, 16 and 64 processors, respectively. Figures 10.A - 10.F and 11.A - 11.F are for the iPSC/860. Figures 12.A - 12.F, 13.A - 13.F, and 14.A - 14.F are for the DELTA. For using 4 processors, as the grid size increases, the computation quickly dominates the communication, especially on the nCUBE 2. The communication, then, grows dramatically as the number of processors increases. Given the number of processors and the problem, the iPSC/860 shows a much too high ratio of communication-to-computation speeds, this ratio is somewhat improved on the DELTA. We also note that the *send* communication is surprisingly costly on the iPSC/860.

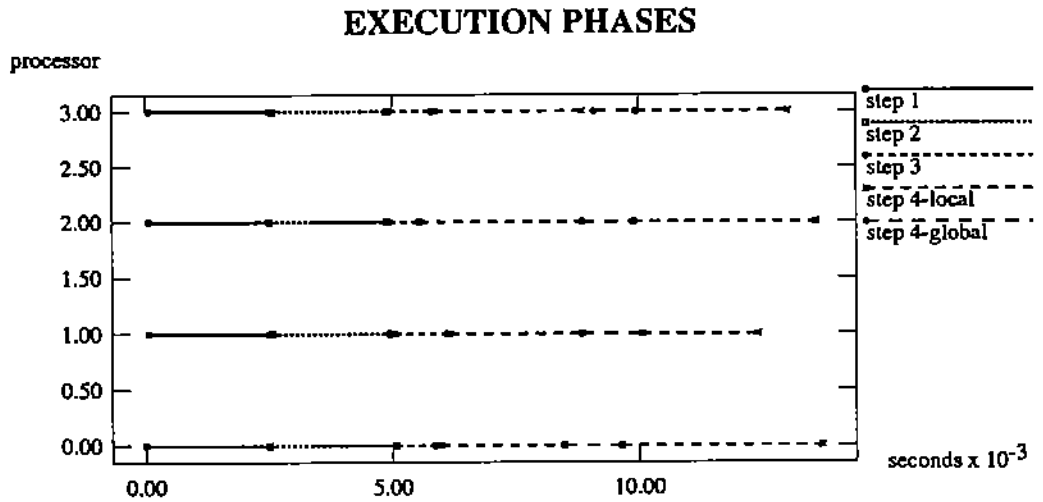

Figure 7.A. 9 × 9 grid problem

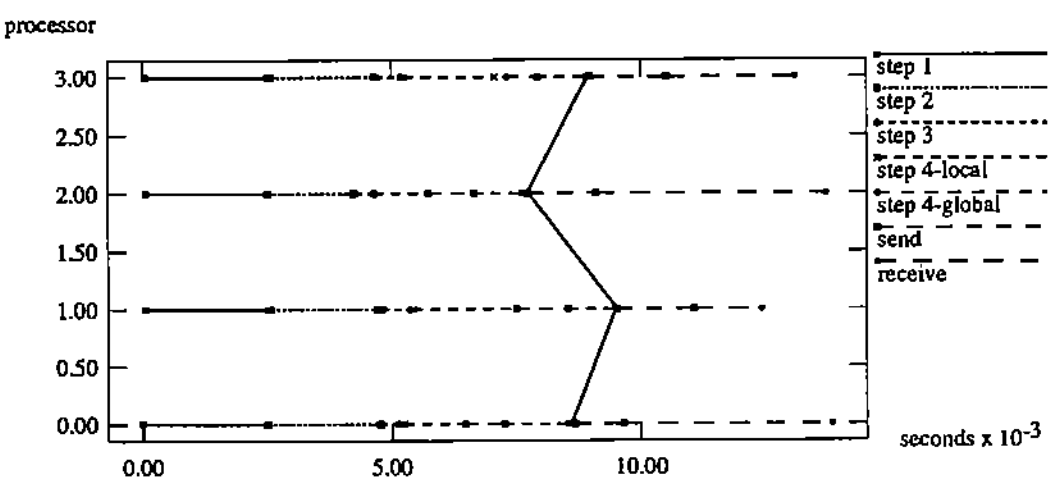

Figure 7.B. 9 × 9 grid problem

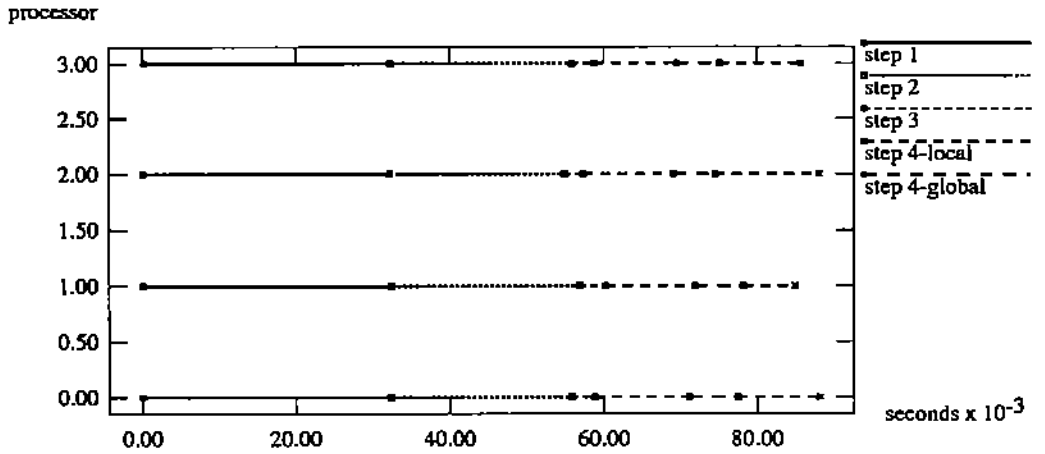

Figure 7.C. 17 × 17 grid problem

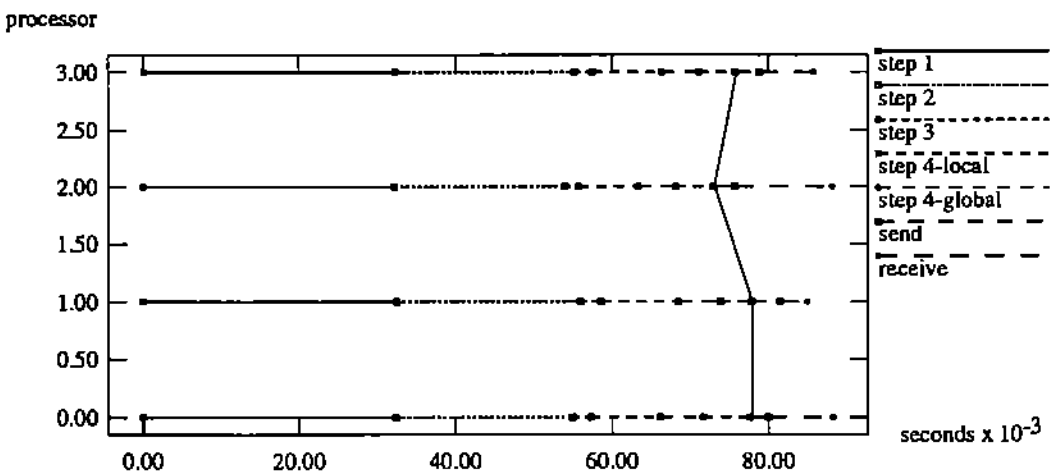

Figure 7.D. 17 × 17 grid problem

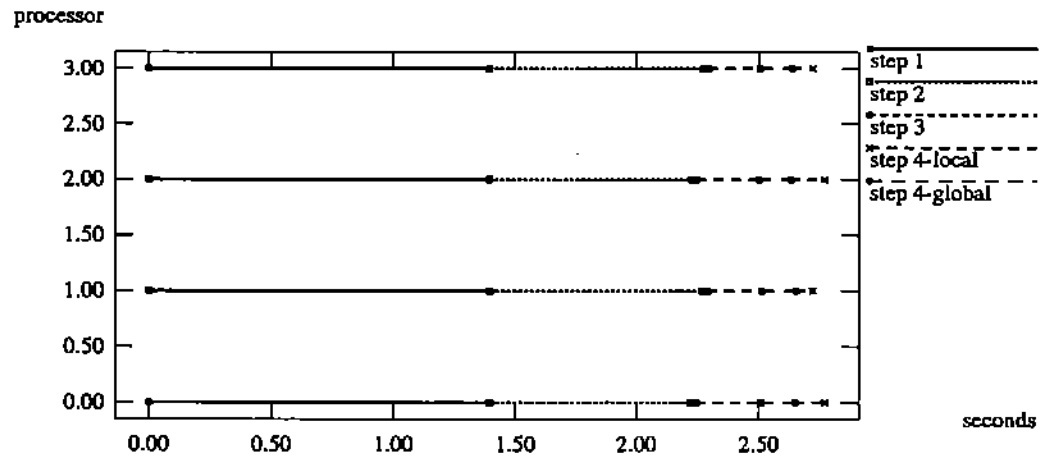

Figure 7.E. 49 × 49 grid problem

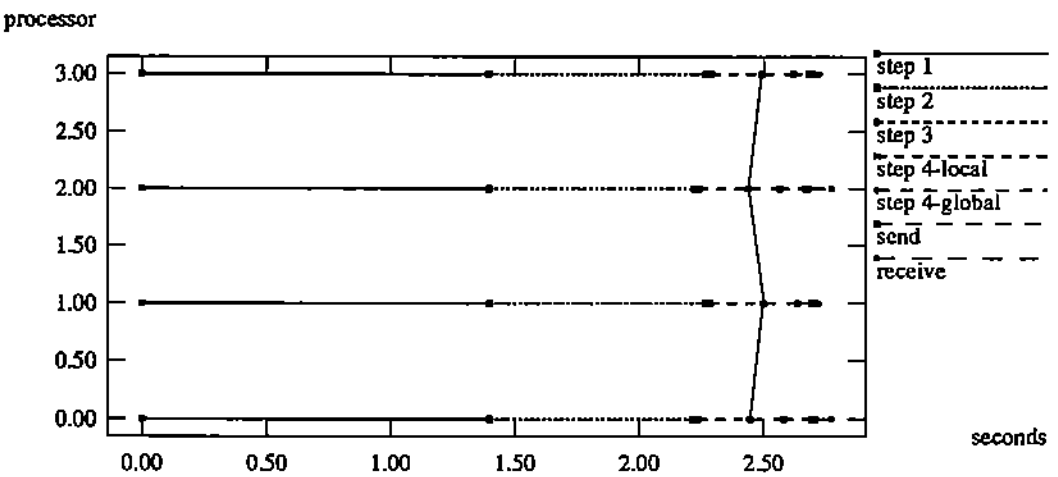

Figure 7.F. 49 × 49 grid problem

Figure 7: Performance visualization on the nCUBE 2 using 16 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

EXECUTION PHASES

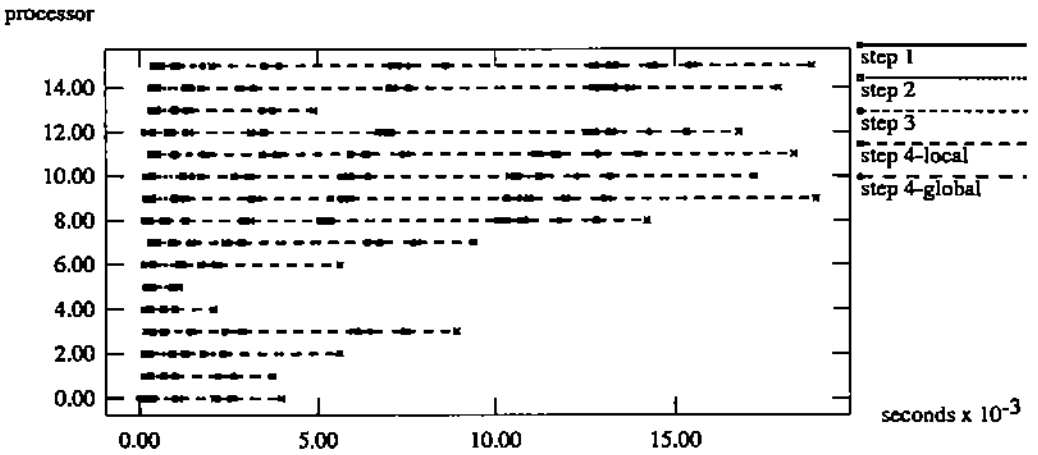

Figure 8.A. 9 × 9 grid problem

COMPUTATION LOAD

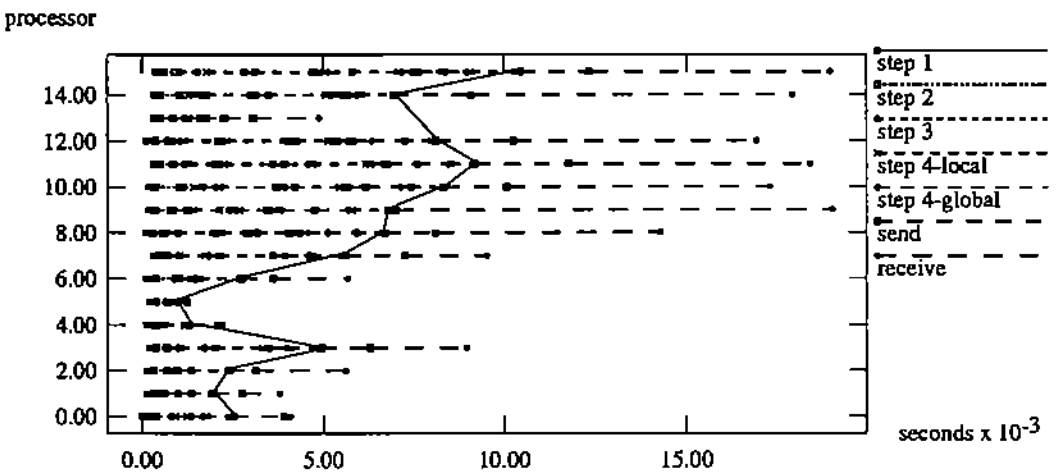

Figure 8.B. 9 × 9 grid problem

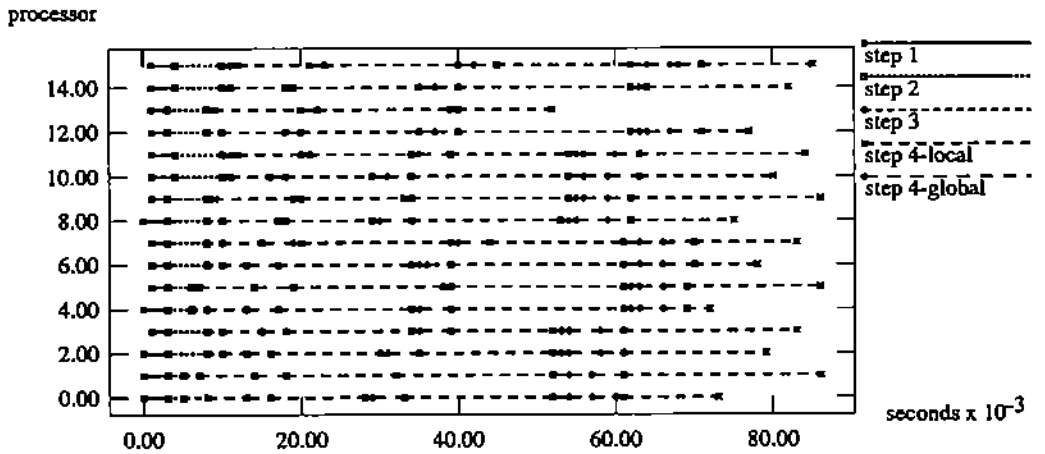

Figure 8.C. 17 × 17 grid problem

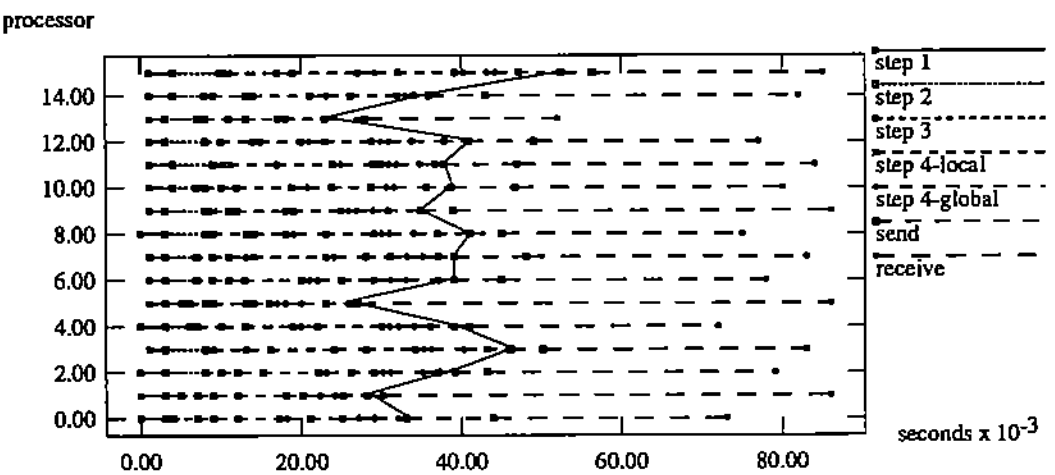

Figure 8.D. 17 × 17 grid problem

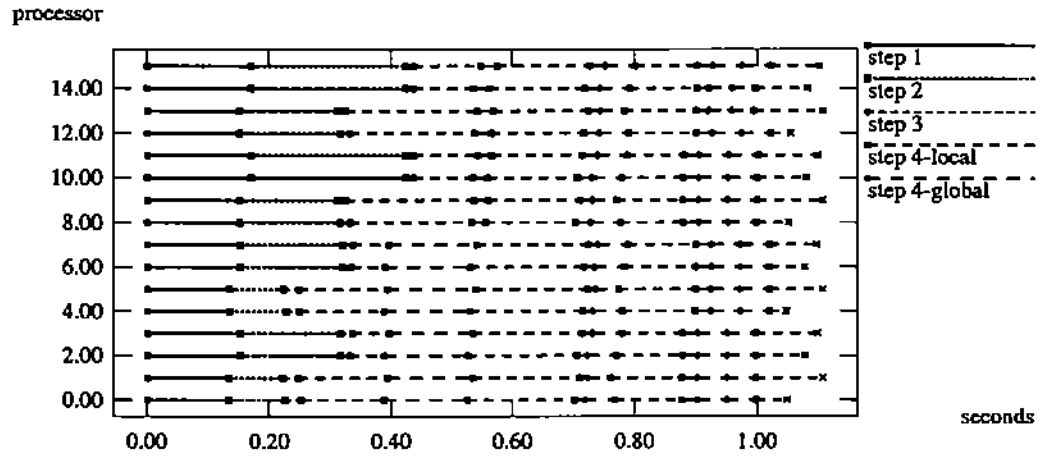

Figure 8.E. 49 × 49 grid problem

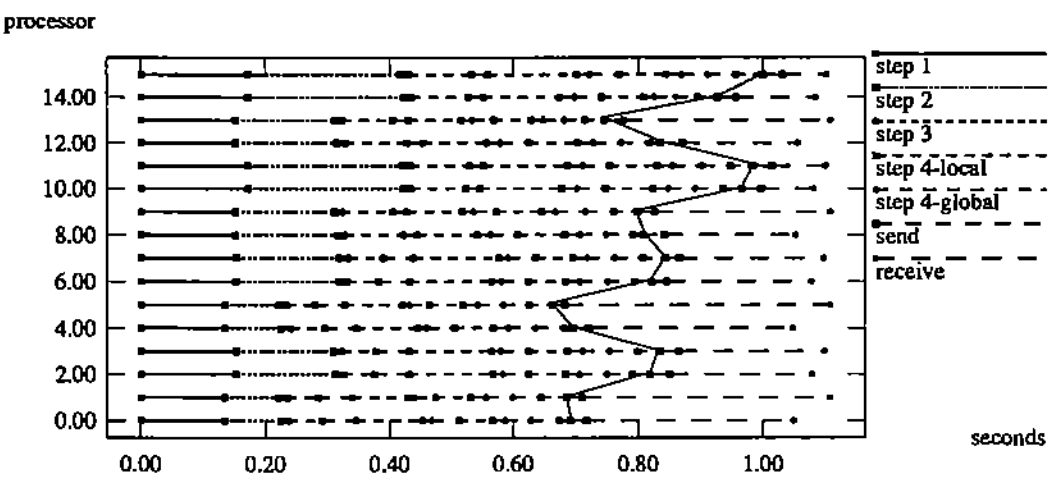

Figure 8.F. 49 × 49 grid problem

Figure 8: Performance visualization on the nCUBE 2 using 16 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

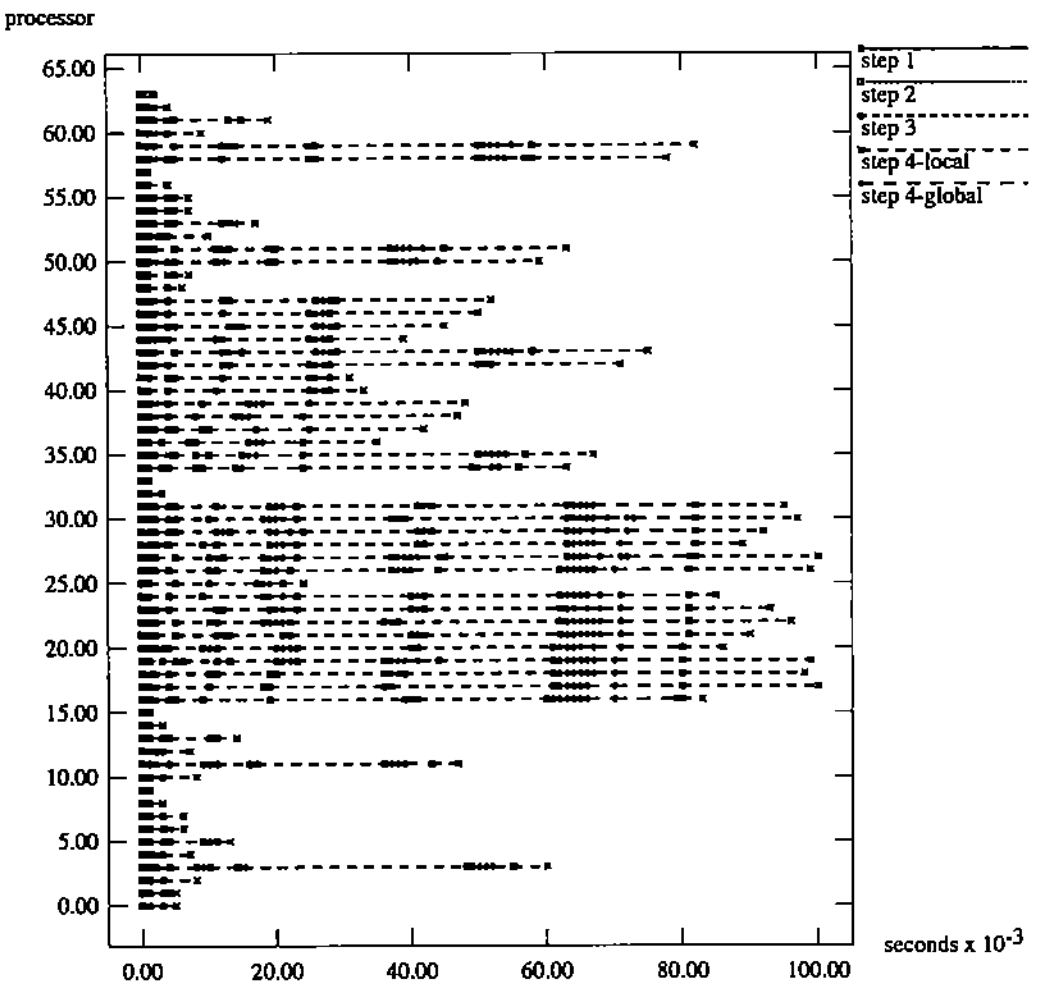

Figure 9.A. 17 × 17 grid problem

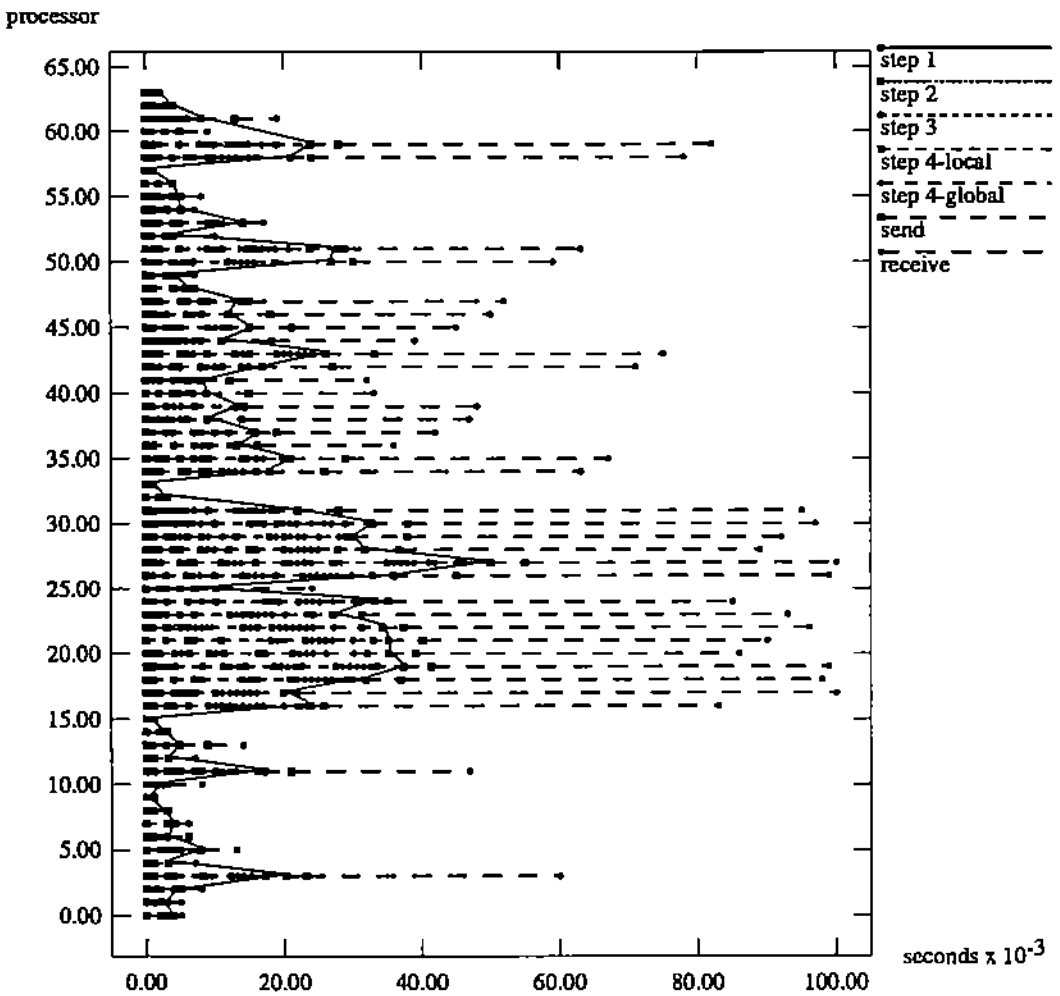

Figure 9.B. 17 x 17 grid problem

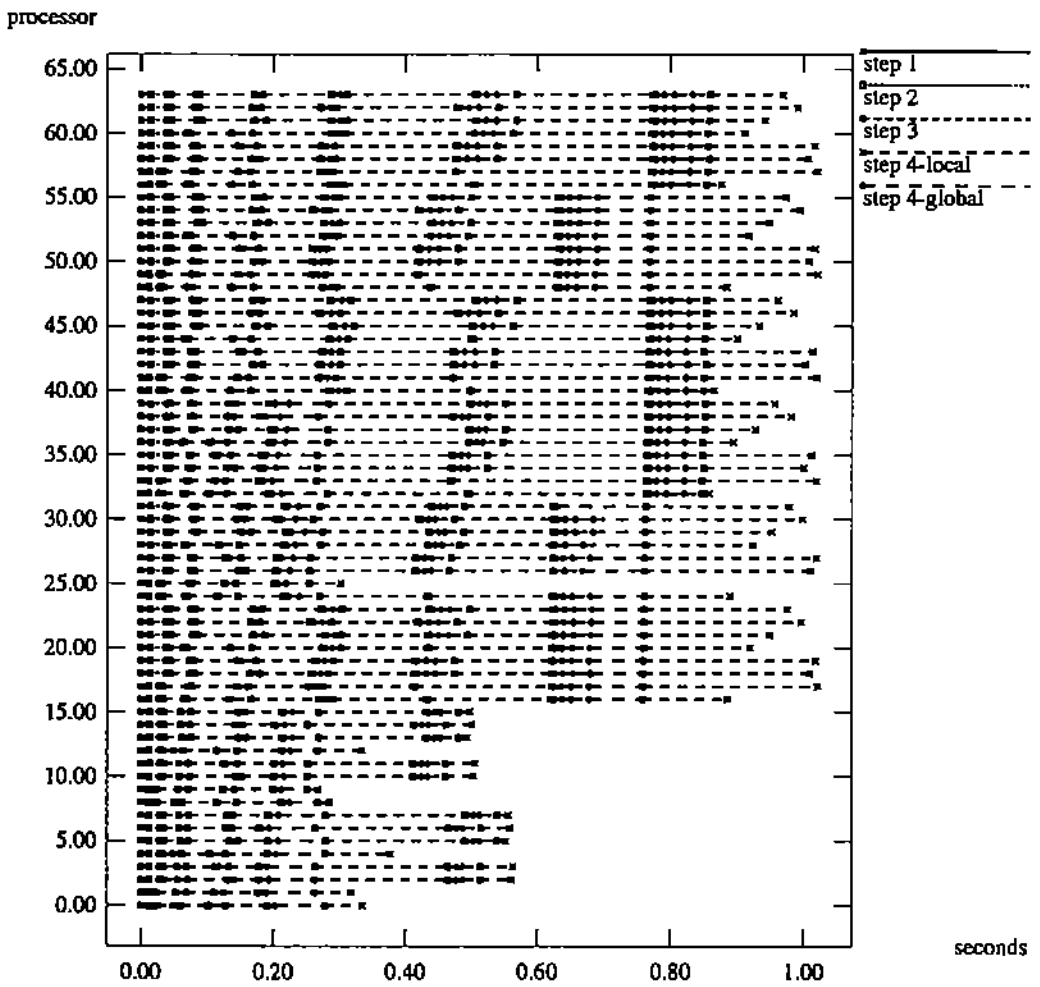

Figure 9.C. 49 × 49 grid problem

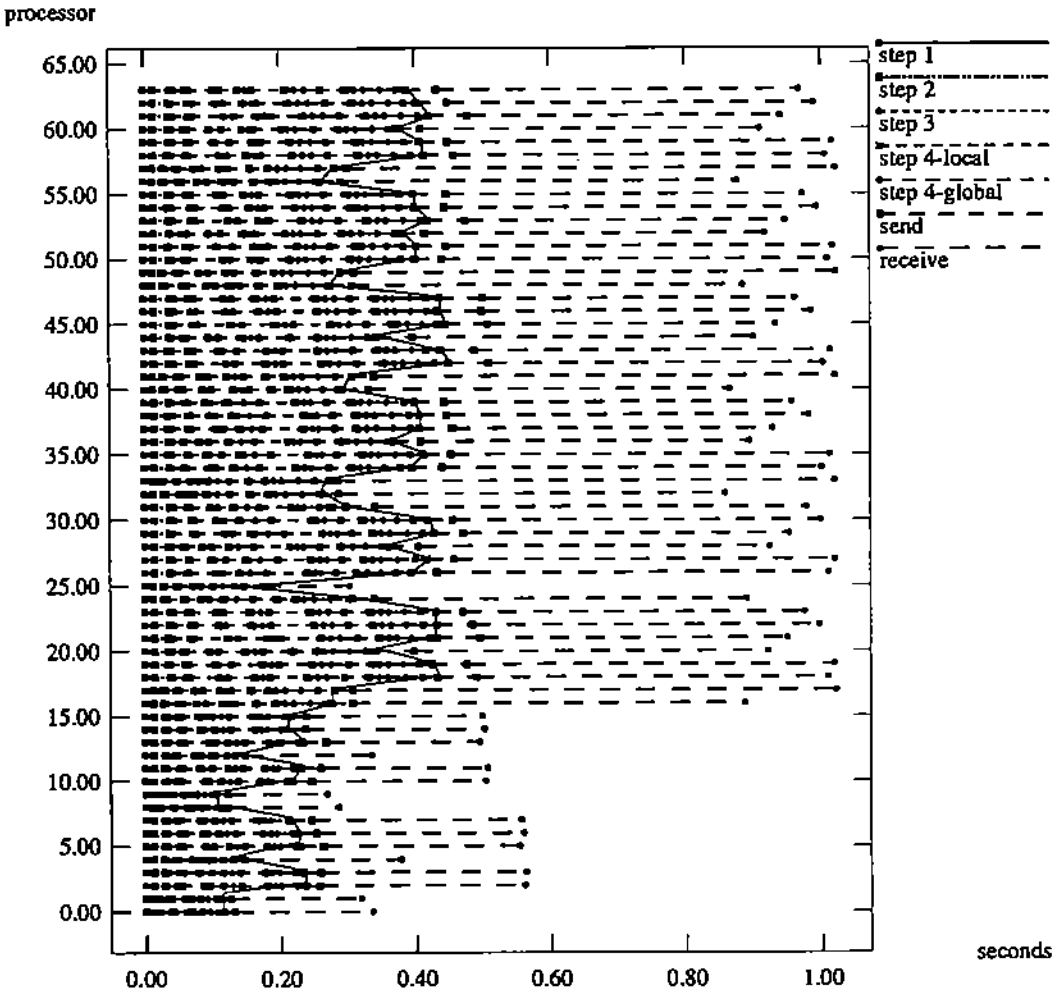

Figure 9.D. 49 x 49 grid problem

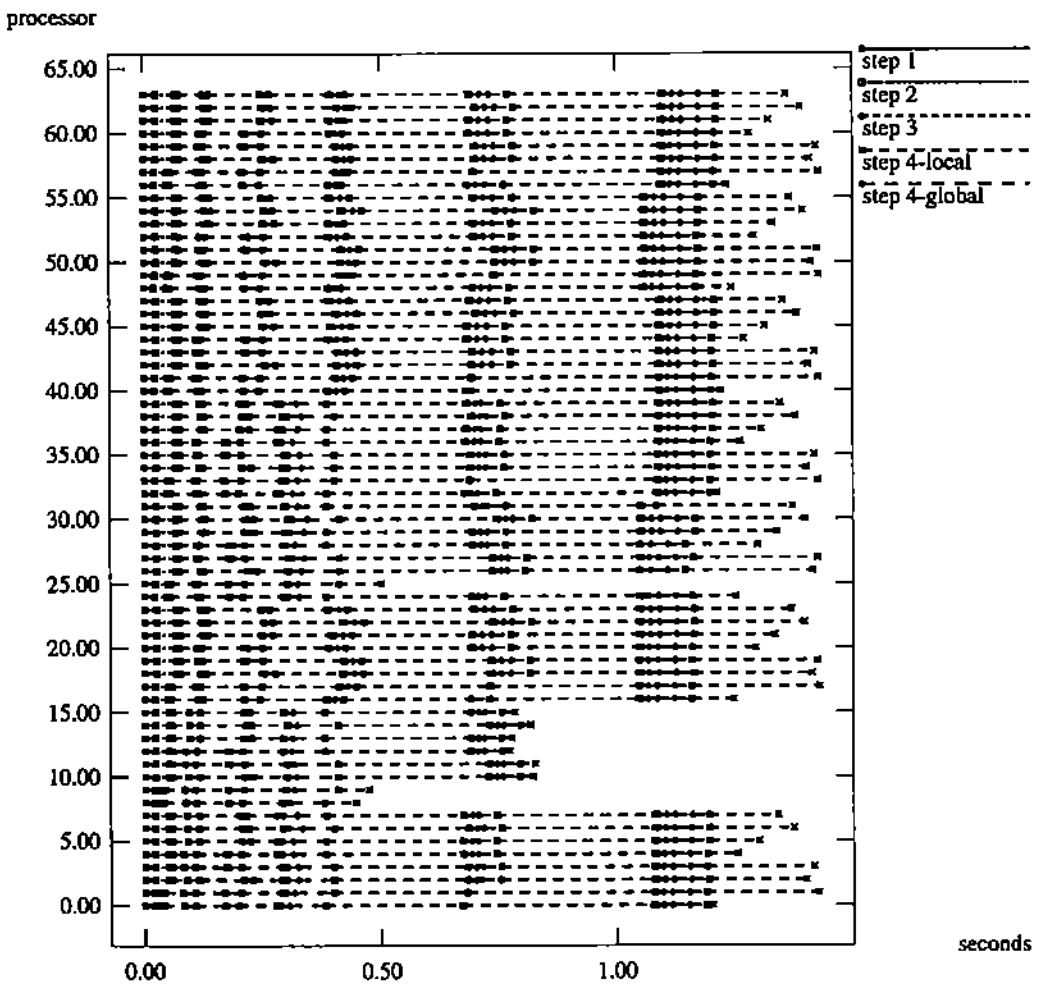

Figure 9.E. 57 × 57 grid problem

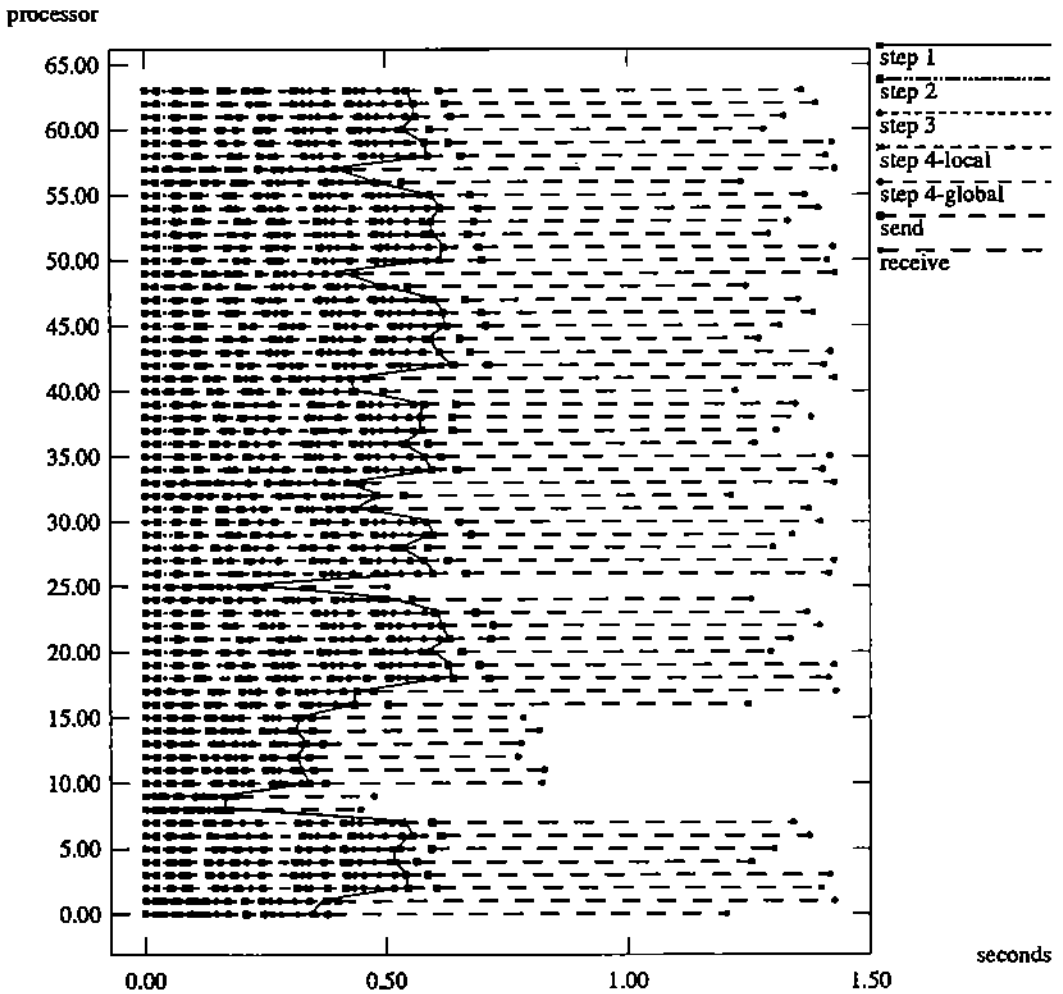

Figure 9.F. 57 × 57 grid problem

Figure 9: Performance visualization on the nCUBE 2 using 64 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

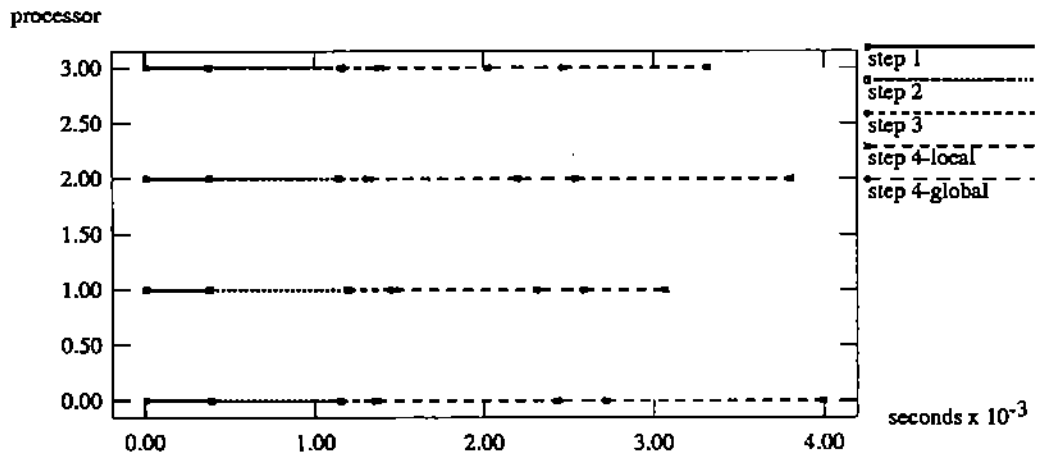

Figure 10.A. 9 × 9 grid problem

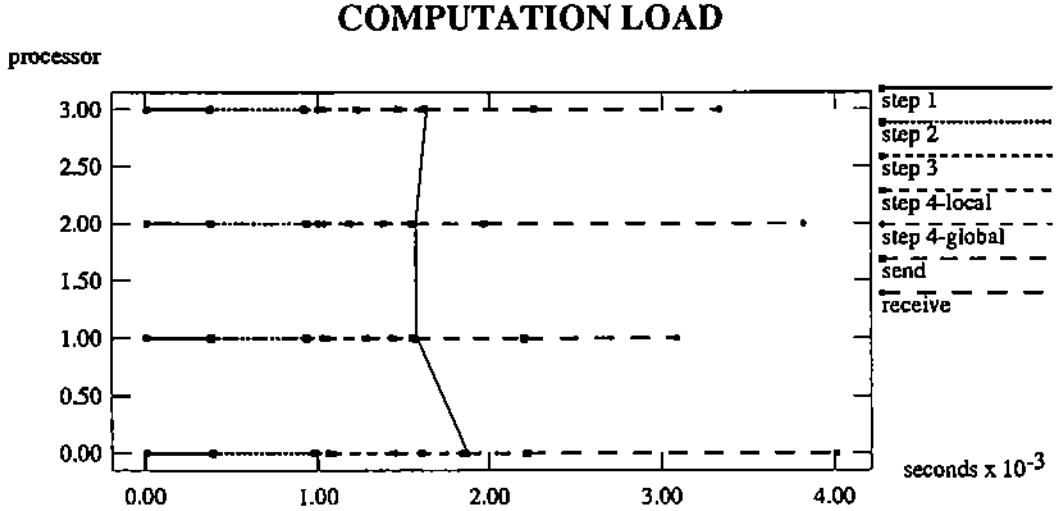

Figure 10.B. 9 × 9 grid problem

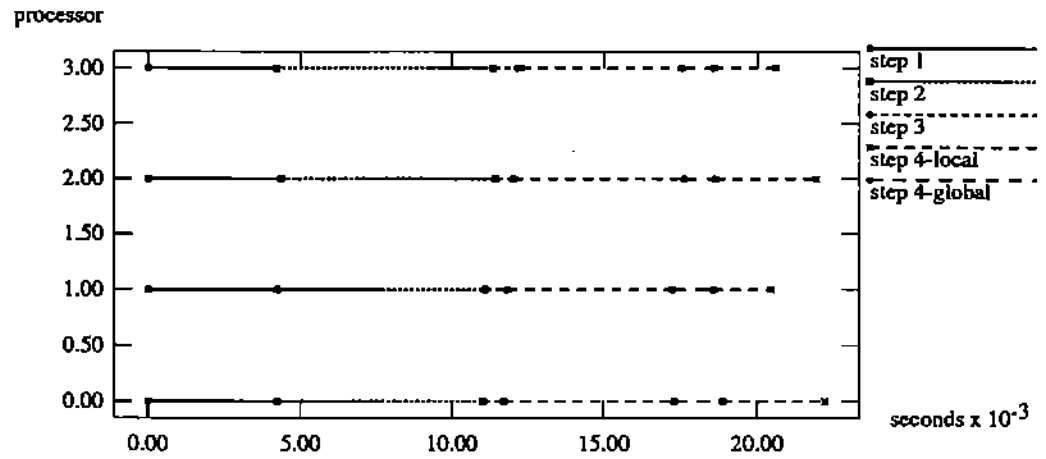

Figure 10.C. 17 × 17 grid problem

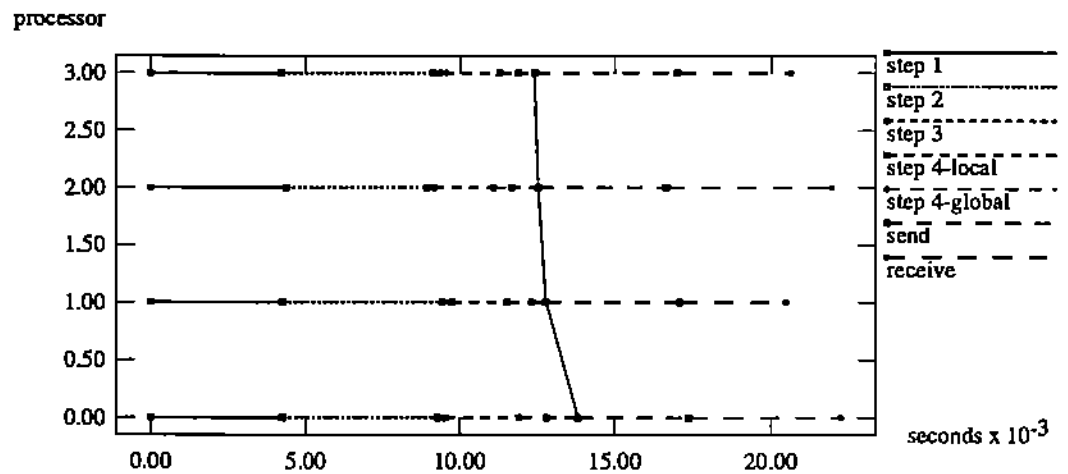

Figure 10.D. 17 × 17 grid problem

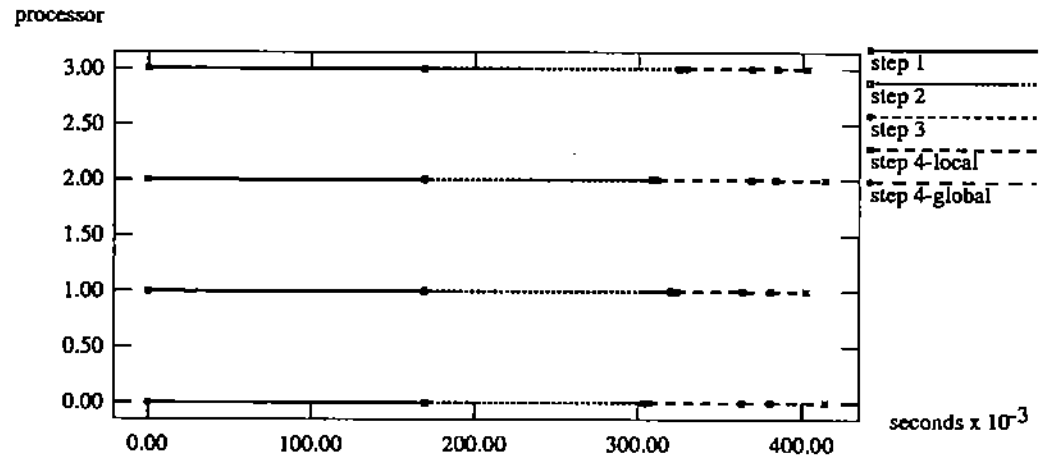

Figure 10.E. 49 × 49 grid problem

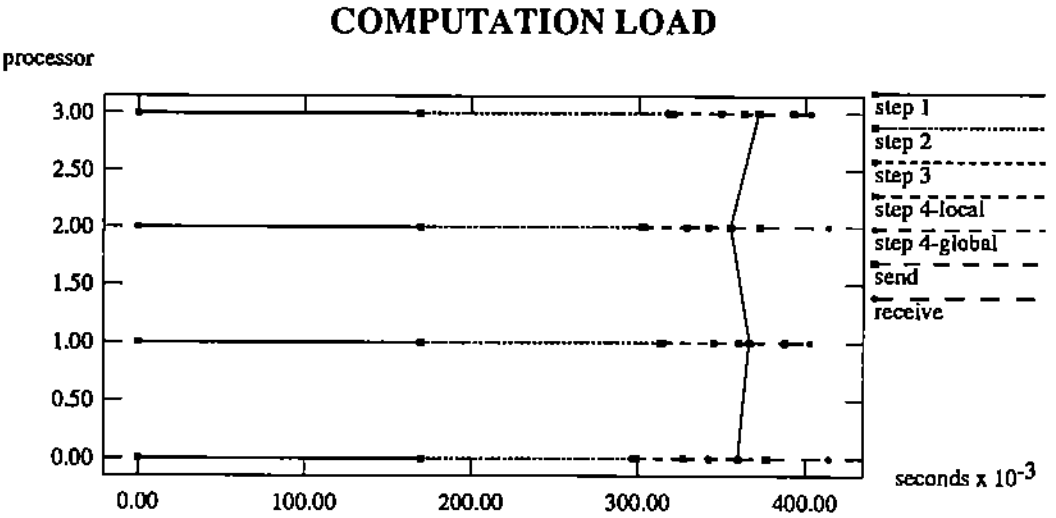

Figure 10.F. 49 × 49 grid problem

Figure 10: Performance visualization on the Intel iPSC/860 using 4 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

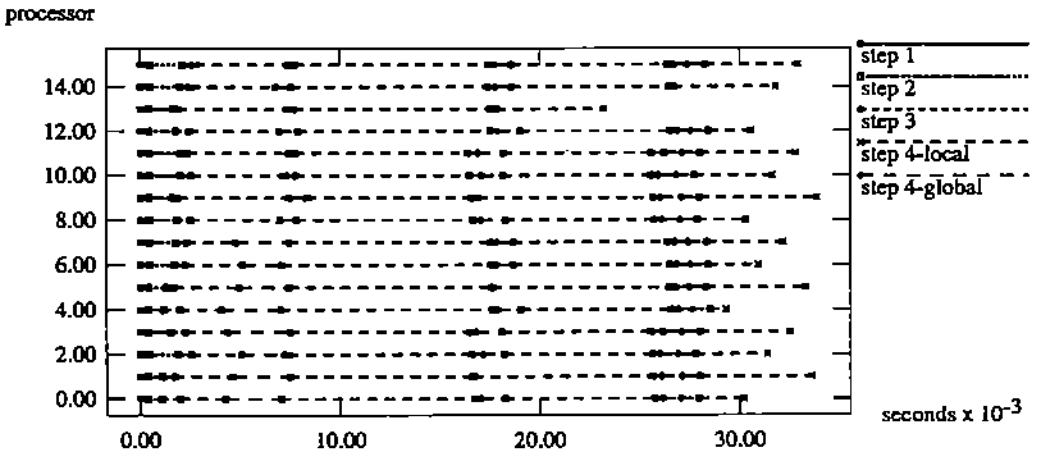

Figure 11.A. 17 × 17 grid problem

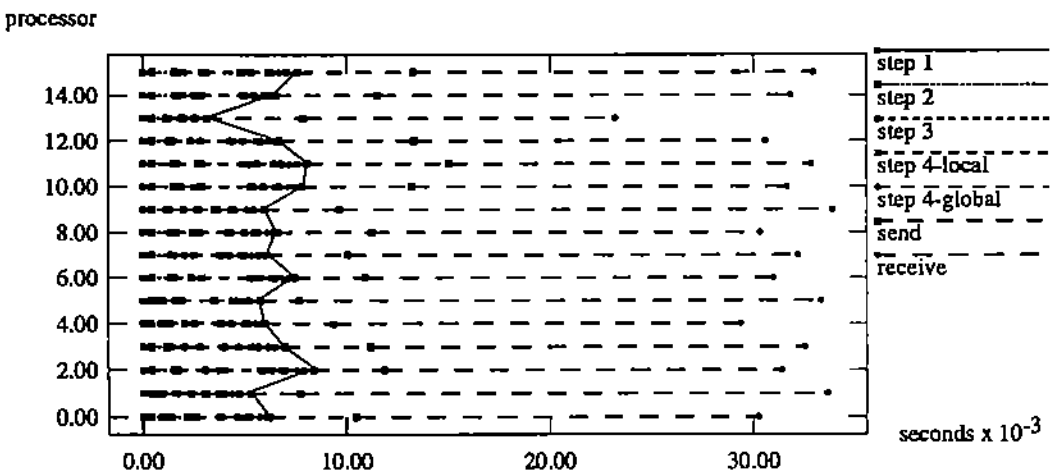

Figure 11.B. 17 × 17 grid problem

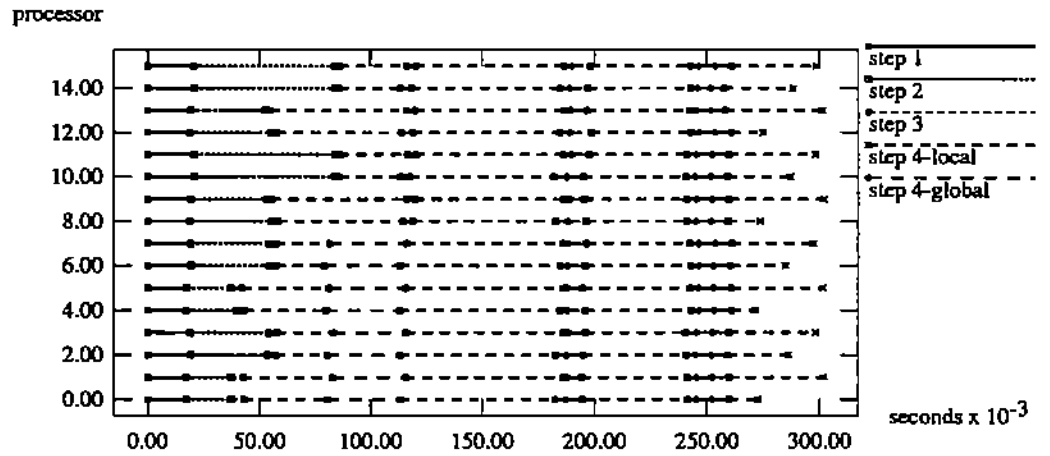

Figure 11.C. 49 × 49 grid problem

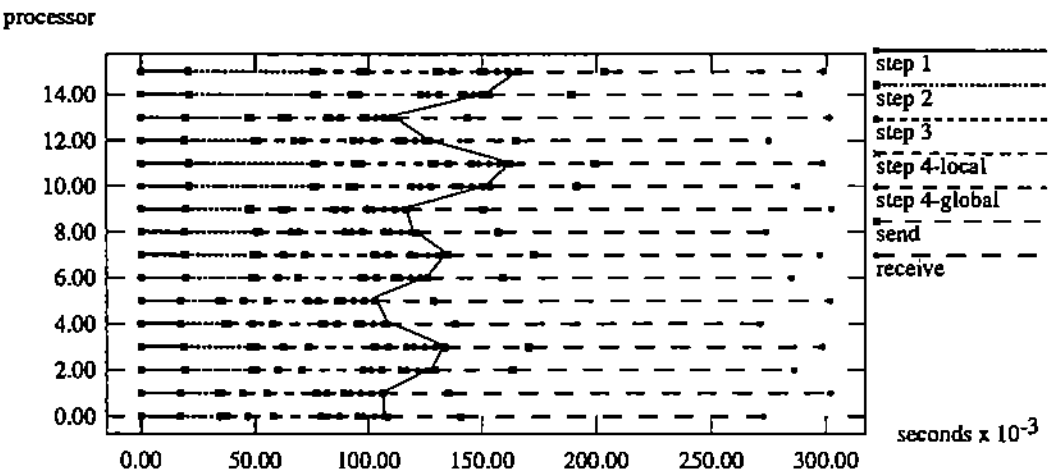

Figure 11.D. 49 × 49 grid problem

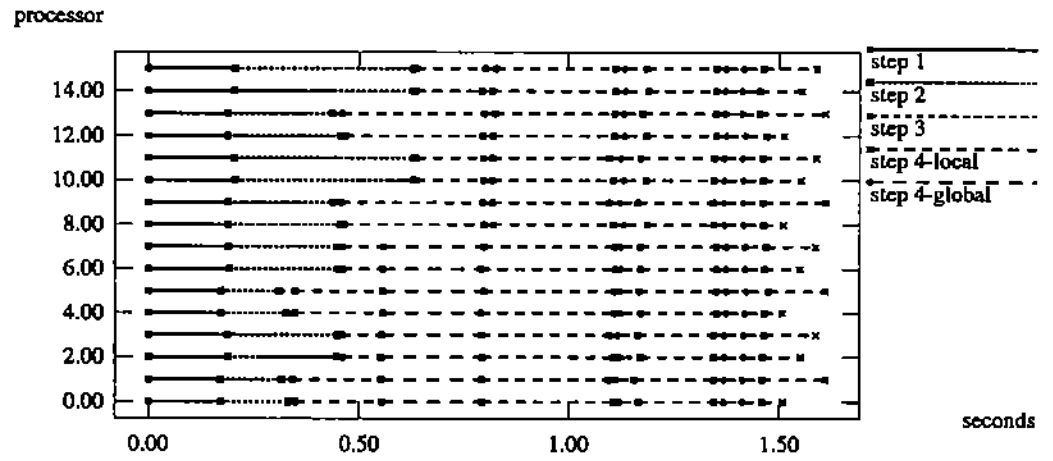

Figure 11.E. 97 × 97 grid problem

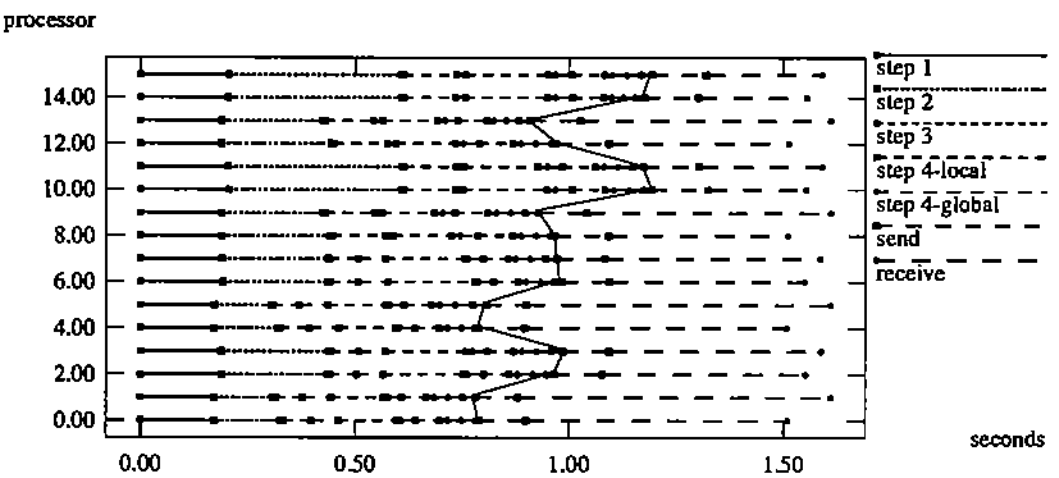

Figure 11.F. 97 × 97 grid problem

Figure 11: Performance visualization on the Intel iPSC/860 using 16 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

EXECUTION PHASES

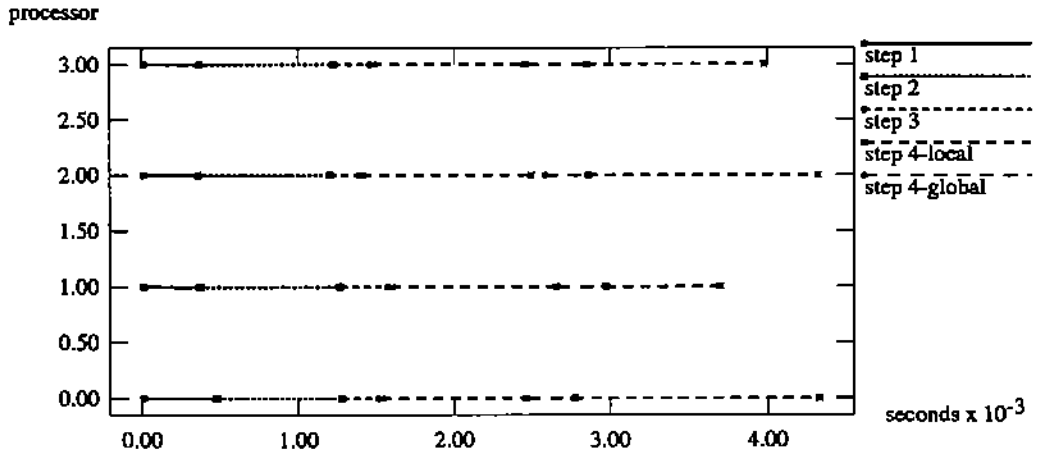

Figure 12.A. 9 × 9 grid problem

COMPUTATION LOAD

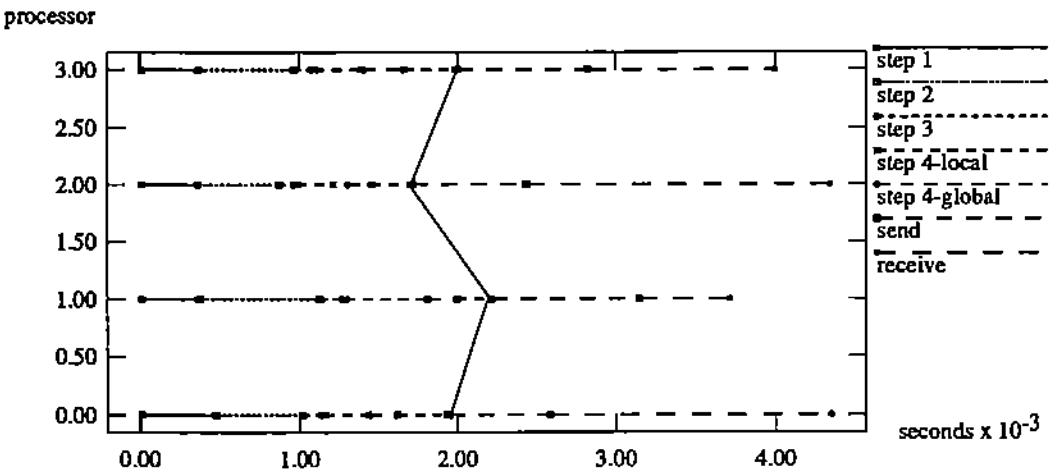

Figure 12.B. 9 × 9 grid problem

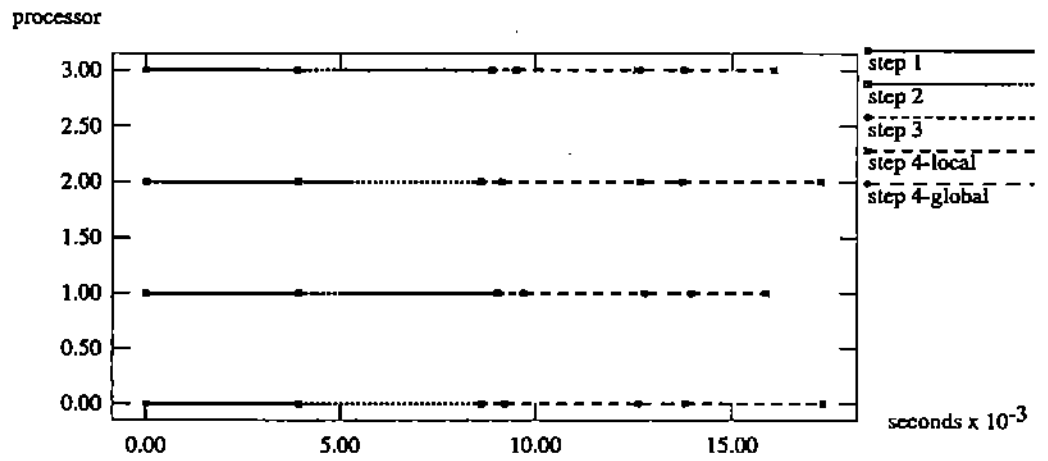

Figure 12.C. 17 × 17 grid problem

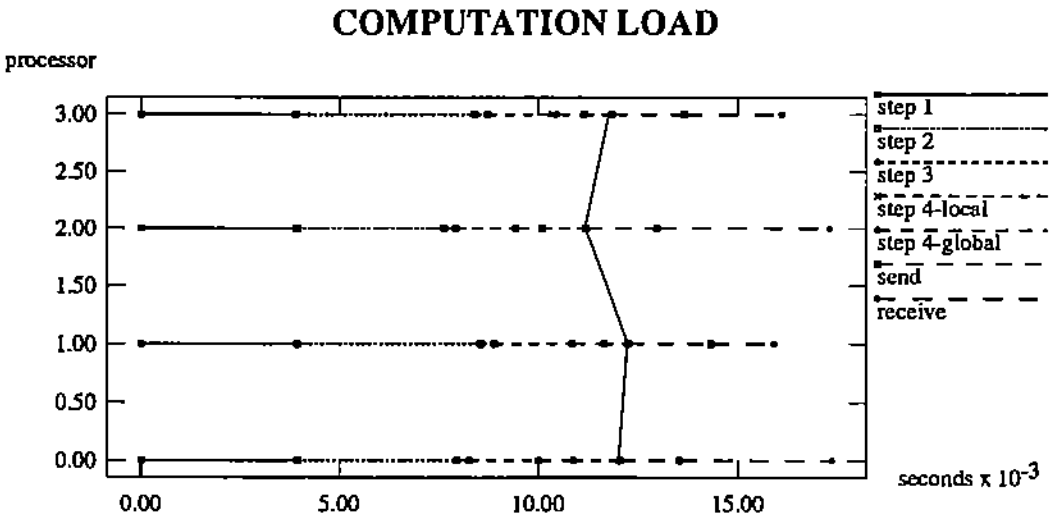

Figure 12.D. 17 × 17 grid problem

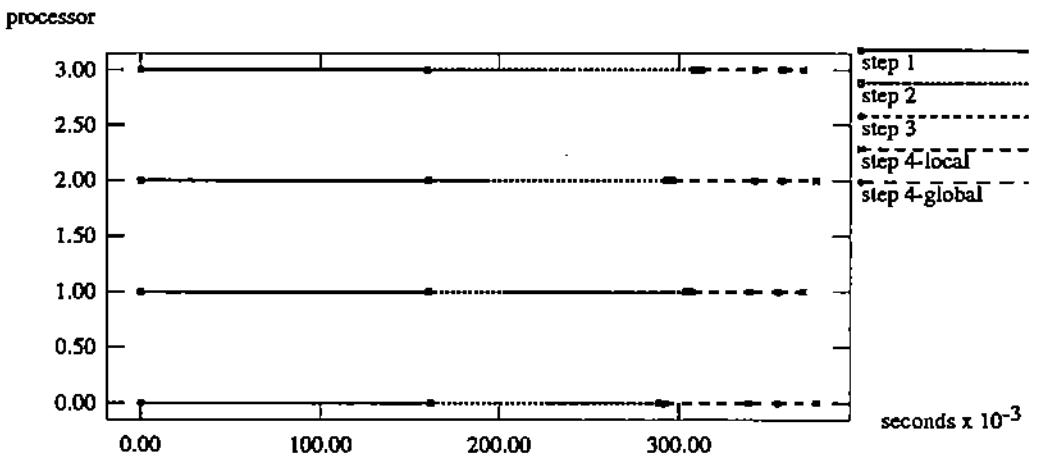

Figure 12.E. 49 × 49 grid problem

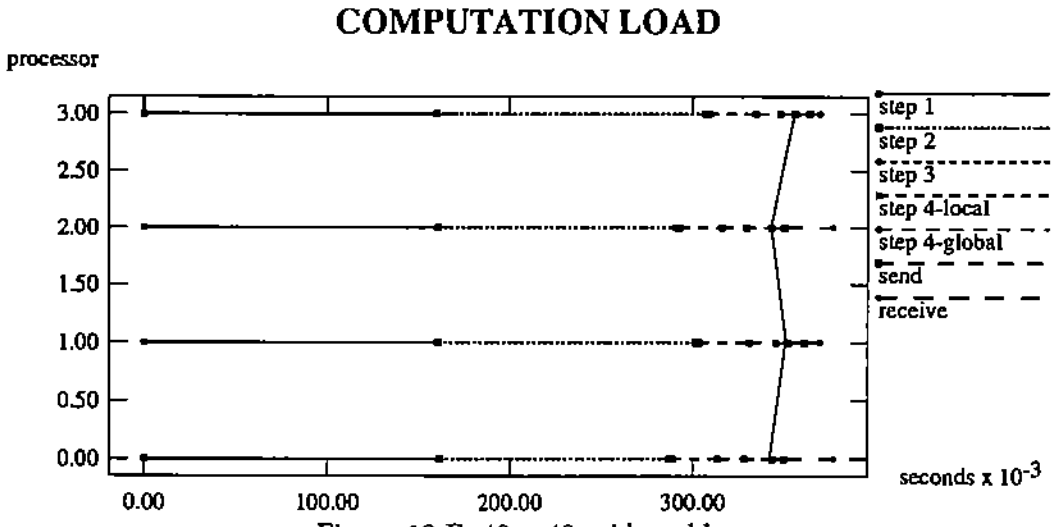

Figure 12.F. 49 × 49 grid problem

Figure 12: Performance visualization on the Intel DELTA using 4 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

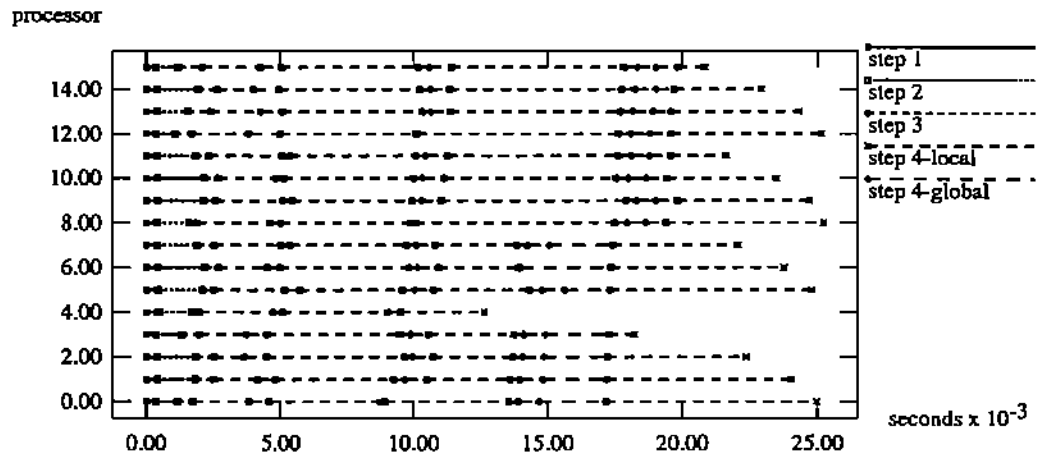

Figure 13.A. 17 × 17 grid problem

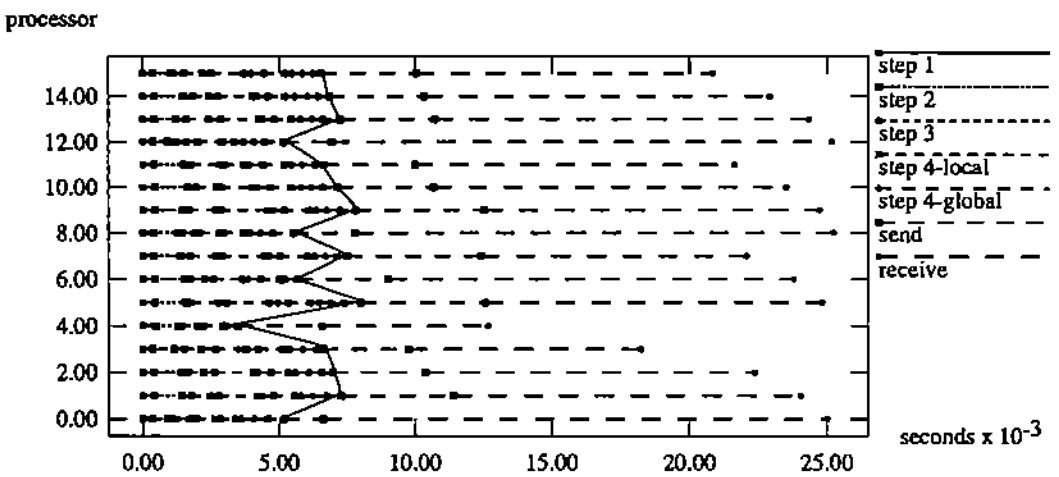

Figure 13.B. 17 × 17 grid problem

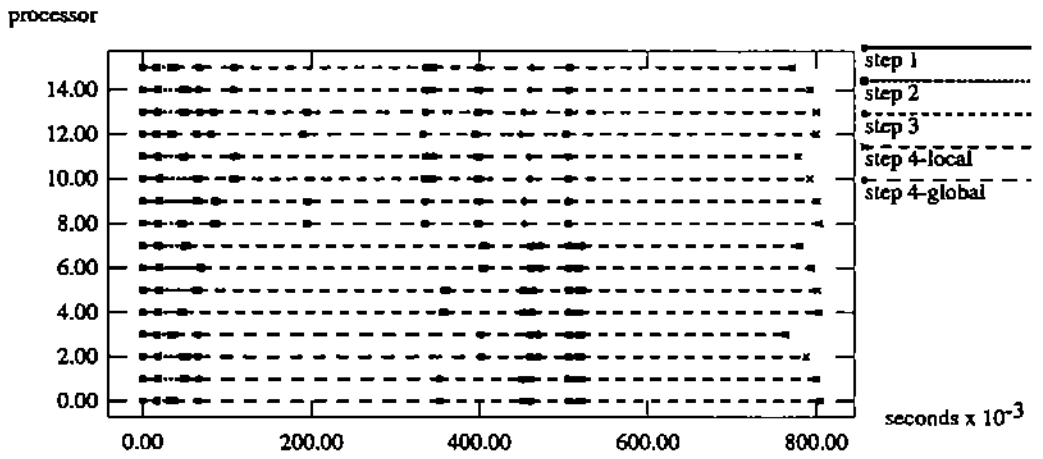

Figure 13.C. 49 × 49 grid problem

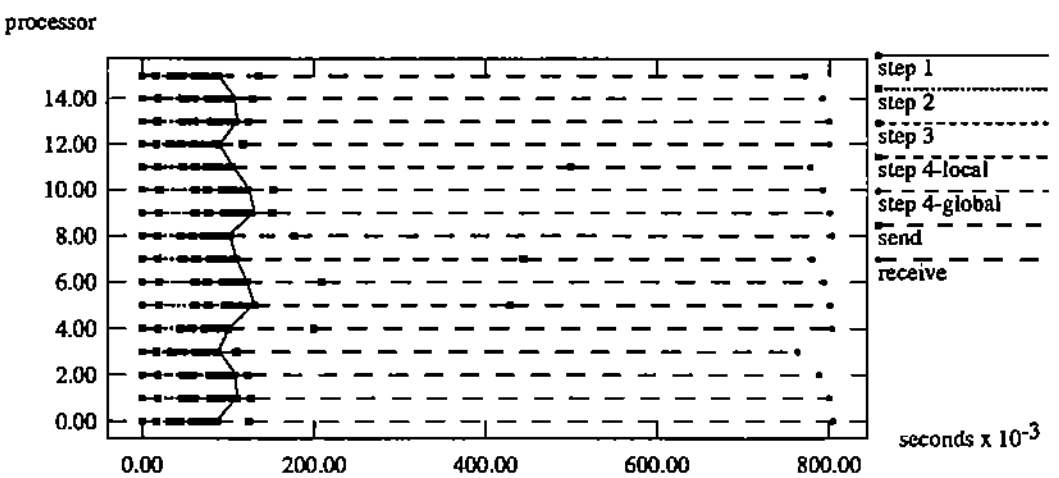

Figure 13.D. 49 × 49 grid problem

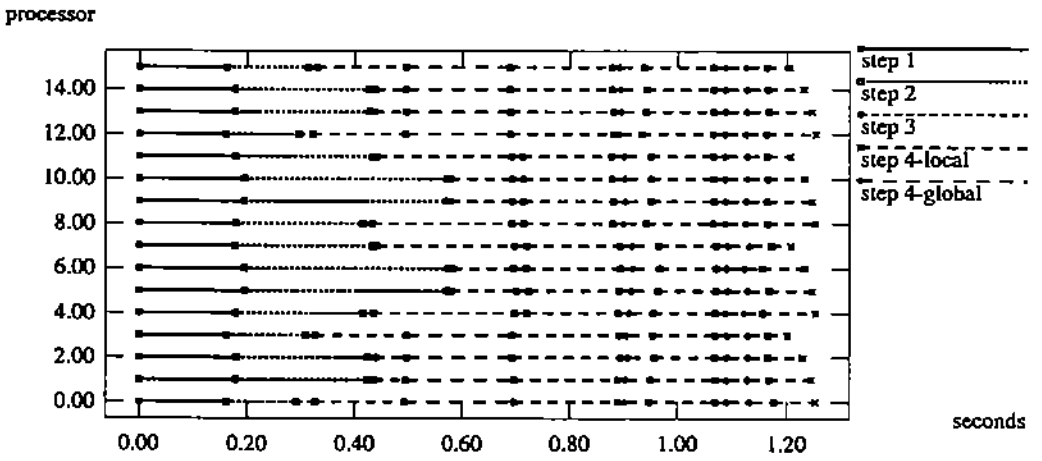

Figure 13.E. 97 × 97 grid problem

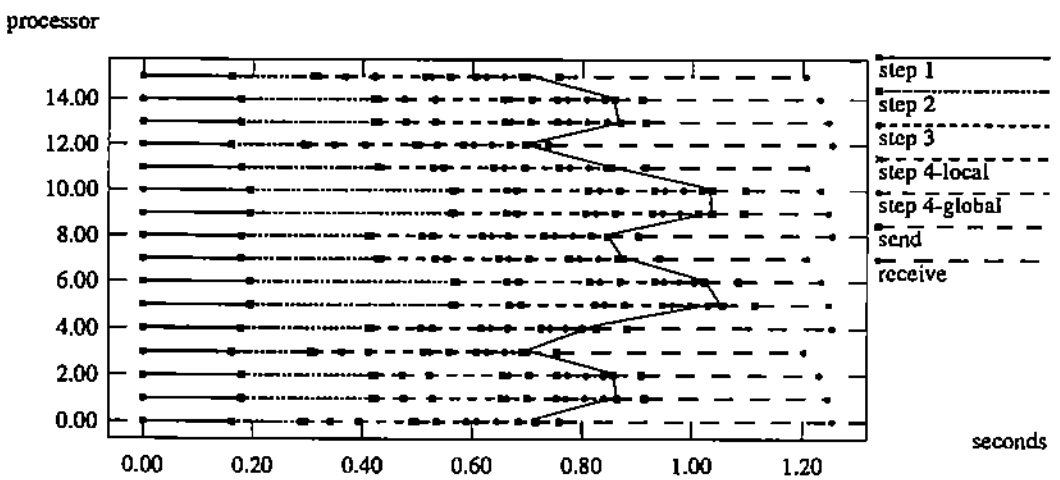

Figure 13.F. 97 × 97 grid problem

Figure 13: Performance visualization on the Intel DELTA using 16 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

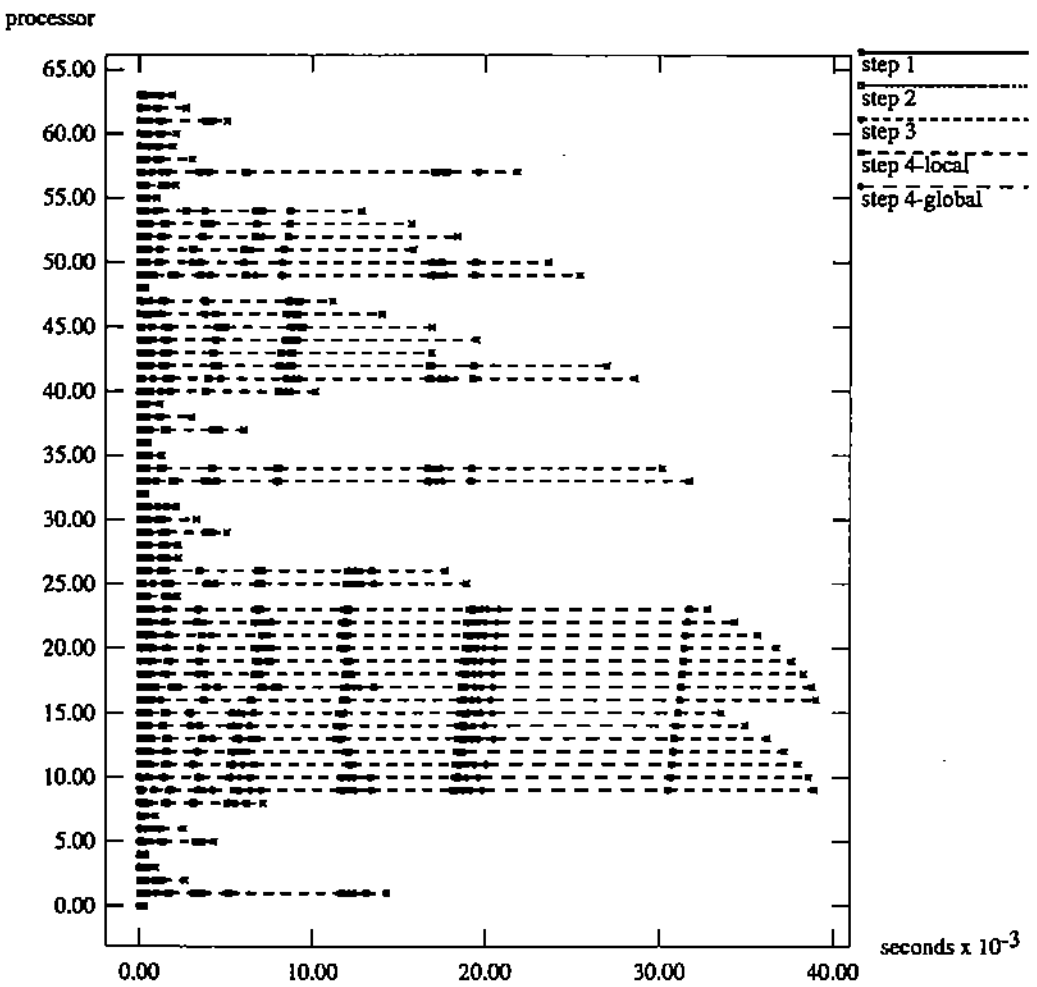

Figure 14.A. 17 × 17 grid problem

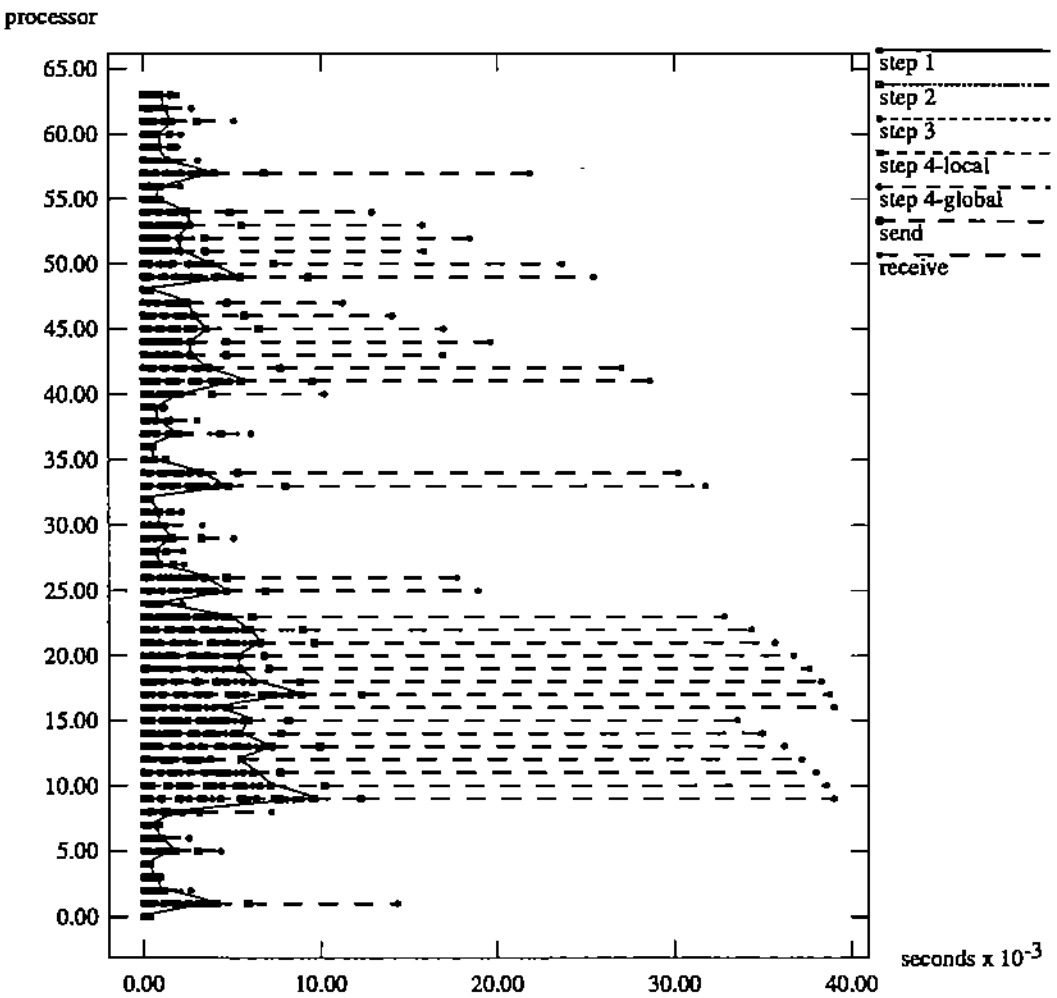

Figure 14.B. 17 × 17 grid problem

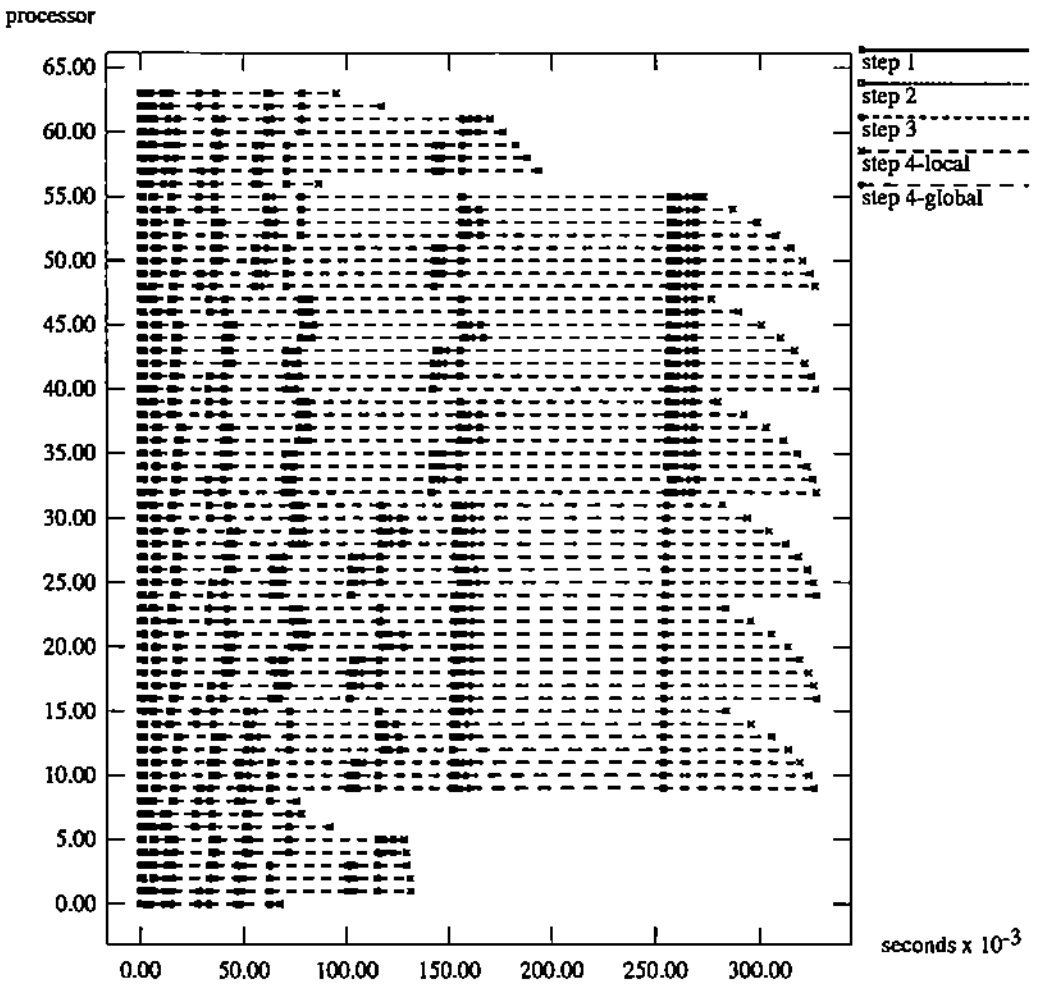

Figure 14.C. 49 × 49 grid problem

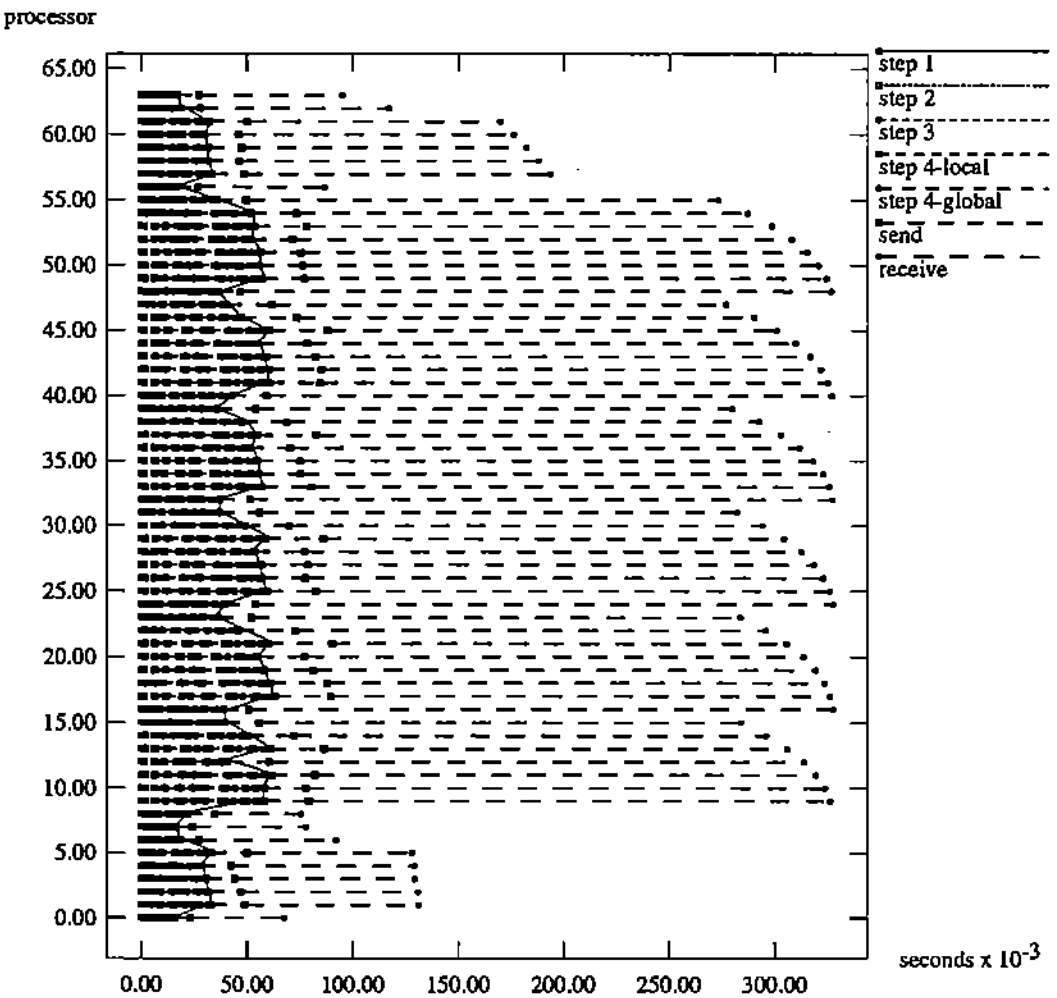

Figure 14.D. 49 × 49 grid problem

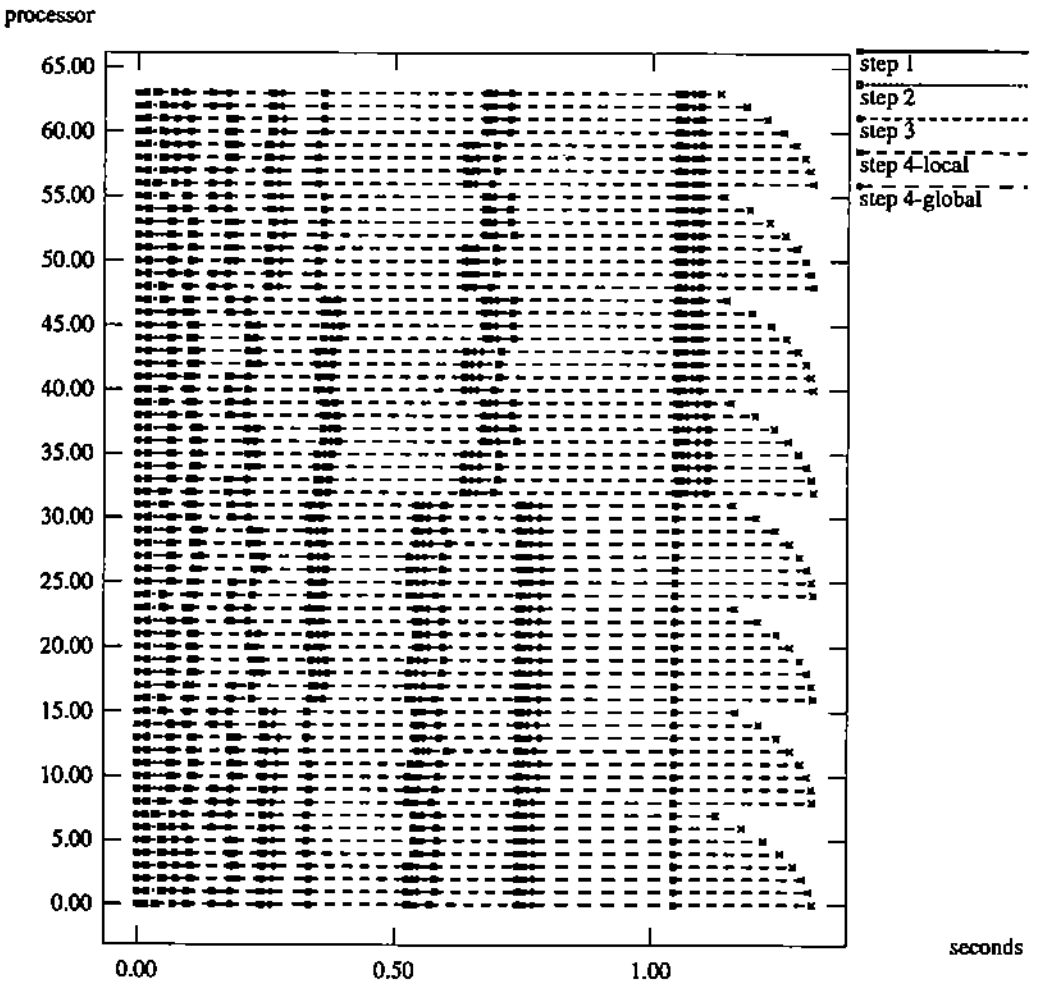

Figure 14.E. 97 × 97 grid problem

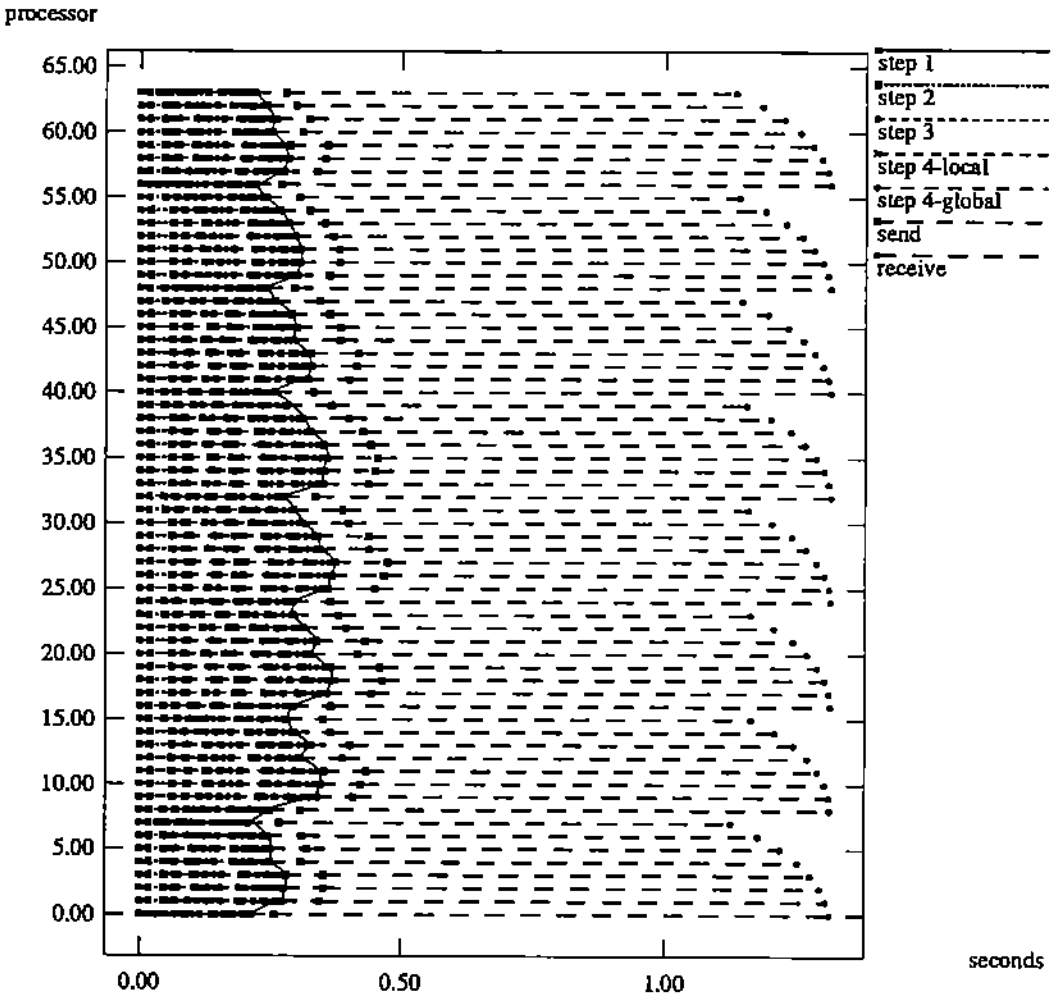

Figure 14.F. 97 × 97 grid problem

Figure 14: Performance visualization on the Intel DELTA using 64 processors. The execution phases correspond to the steps of the sparse Gauss elimination. The computational load has the send/receive times extracted and moved to the right of the vertical lines.

Another important characteristic seen from these pictures is that the load balances are very different for various domain decompositions. This is because the number of connected interface unknowns varies from subdomain to subdomain. We discuss several ways to optimize the performance according to characteristics of the machine and the problem in [7] and show how to obtain substantial improvements. These optimizations include adaptations in assignment, scheduling, message packing, pipelining, and so on.

4 Conclusions

This paper provides extensive data for the performance of a PDE sparse solver benchmark using three popular MIMD multiprocessors. It is shown that these supercomputers are still very far away from being well utilized for this benchmark, especially for a large number of processors. Therefore, one has to very carefully optimize the performance adapting to the characteristics of a given machine and problem. We believe that it is almost impossible to make full use of the power provided by these computers to solve these PDE problems using sparse matrix techniques because of the small grain sparsity presented by the problems and the large granularity preferred by machines. We will further investigate the machine performance for solving PDEs by using iterative methods. It is also of interest to compare the performance of the band matrix LINPACK solver for these problems. In addition, it is important to see how to efficiently solve three dimensional PDE problems on these machines.

References

[1] Dongarra, J., (1992), "Performance of Various Computers Using Standard Linear Equations Software", Technical Report, University of Tennessee, CS-89-85, March, 1992.

[2] Dunigan, T., (1991), "Performance of the Intel iPSC/860 and Ncube 6400 Hypercubes", ORNL/TM-11491, Oak Ridge National Laboratories, Oak Ridge, TN.

[3] Dunigan, T., (1992), "Communication Performance of the Intel Touchstone DELTA Mesh", ORNL/TM-11983, Oak Ridge National Laboratories, Oak Ridge, TN.

[4] Eisenstat, S.C., M.C. Gursky, M.H. Schultz, and A.H. Sherman, (1977), "Yale Sparse Matrix Package, II: The Nonsymmetric Codes", Report 114, Computer Science, Yale University.

[5] Mu, M., and J. Rice, (1990), "Parallel Sparse: Data Structure and Algorithm", CSD-TR-974, CER-90-17, Computer Science Department, Purdue University.

[6] Mu, M., and J. Rice, (1990), "A New Organization of Sparse Gauss Elimination for Solving PDEs", CSD-TR-991, CER-90-22, Computer Science Department, Purdue University.

[7] Mu, M. and J. Rice, (1991), "Unstructured Scheduling in Parallel PDE Sparse Solvers on Distributed Memory Machines", presented at the *Tenth Parallel Circus*, Oak Ridge, TN, October 25, 1991, CSD-TR-91-077, CER-91-40, Computer Science Department, Purdue University.

[8] Mu, M. and J. Rice, (1992), "Performance of PDE Sparse Solvers on Hypercubes", *Unstructured Scientific Computation on Multiprocessors*, (P. Mehrotra, J. Saltz and R. Voigt, eds.), MIT Press.

[9] Mu, M. and J. Rice, (1992), "A Grid Based Subtree-Subcube Assignment Strategy for Solving PDEs on Hypercubes", *SIAM J. Sci. Stat. Comput.*, May.

[10] Mu, M. and J. Rice, (1992), "Row Oriented Gauss Elimination on Distributed Memory Multiprocessors", to appear in *Intern. J. High Speed Computing.*

Appendix Experimental Timing Data

Table 3. Experimental timing data (in seconds) for various grids and various machines. For the missing data in the table, "–" means that the grid is not compatible with the domain decomposition; blank spaces are for the data unavailable due to the memory limitation.

	nCUBE 2			
Grid Size	Sequential	4 processors	16 processors	64 processors
5 × 5	1.47E-3	1.95E-3	–	–
7 × 7	6.90E-3	5.71E-3	–	–
9 × 9	2.21E-2	1.20E-2	1.36E-2	–
11 × 11	4.77E-2	2.00E-2	–	–
13 × 13	9.51E-2	3.41E-2	3.30E-2	–
15 × 15	1.60E-1	5.47E-2	–	–
17 × 17	2.63E-1	8.29E-2	6.59E-2	7.30E-2
19 × 19	3.77E-1	1.22E-1	–	–
21 × 21	5.63E-1	1.67E-1	1.14E-1	–
23 × 23	7.57E-1	2.24E-1	–	–
25 × 25	1.04	2.99E-1	1.73E-1	1.64E-1
27 × 27	1.31	3.83E-1	–	–
29 × 29	1.73	4.89E-1	2.45E-1	–
31 × 31	2.14	6.03E-1	–	–
33 × 33	2.70	7.54E-1	3.40E-1	3.19E-1
35 × 35	3.19	9.04E-1	–	–
37 × 37	3.95	1.09	4.89E-1	–
39 × 39	4.66	1.28	–	–
41 × 41	5.62	1.55	6.59E-1	5.48E-1
43 × 43	6.42	1.77	–	–
45 × 45	7.58	2.08	8.43E-1	–
47 × 47	8.66	2.37	–	–
49 × 49	1.01E+1	2.76	1.07	8.87E-1
51 × 51	1.12E+1	3.08	–	–
53 × 53	1.30E+1	3.53	1.40	–
55 × 55	1.45E+1	3.94	–	–
57 × 57	1.65E+1	4.68	1.75	1.25
59 × 59	1.82E+1	4.95	–	–
61 × 61	2.05E+1	5.56	2.09	–
63 × 63	2.26E+1	6.12	–	–
65 × 65	2.53E+1	6.85	2.46	1.79
67 × 67	2.74E+1	7.43	–	–
69 × 69	3.04E+1	8.23	3.07	–
71 × 71	3.32E+1	8.98	–	–

Table 3. (continuted) Experimental timing data (in seconds) for various grids and various machines. For the missing data in the table, "−" means that the grid is not compatible with the domain decomposition; blank spaces are for the data unavailable due to the memory limitation.

	nCUBE 2			
Grid Size	Sequential	4 processors	16 processors	64 processors
73 × 73	3.68E+1	9.92	3.65	
75 × 75			−	−
77 × 77				−
79 × 79			−	−
81 × 81				
83 × 83			−	−
85 × 85				−
87 × 87			−	−
89 × 89				
91 × 91			−	−
93 × 93				−
95 × 95			−	−
97 × 97				
99 × 99			−	−
101 × 101				−
103 × 103			−	−
105 × 105				

Table 3. (continuted) Experimental timing data (in seconds) for various grids and various machines. For the missing data in the table, "–" means that the grid is not compatible with the domain decomposition; blank spaces are for the data unavailable due to the memory limitation.

	iPSC/860			DELTA		
Grid Size	Sequential	4 processors	16 processors	4 processors	16 processors	64 processors
5 × 5	2.68E-4	1.11E-3	–	1.03E-3	–	–
7 × 7	1.04E-3	2.09E-3	–	2.06E-3	–	–
9 × 9	3.27E-3	3.36E-3	6.15E-3	4.50E-3	5.39E-3	–
11 × 11	6.59E-3	5.13E-3	–	6.34E-3	–	–
13 × 13	1.27E-2	8.64E-3	1.44E-2	9.22E-3	1.08E-2	–
15 × 15	2.10E-2	1.37E-2	–	1.27E-2	–	–
17 × 17	3.46E-2	2.17e-2	3.21E-2	1.78E-2	1.95E-2	4.59E-2
19 × 19	4.82E-2	2.77E-2	–	2.25E-2	–	–
21 × 21	7.19E-2	3.55E-2	5.09E-2	3.05E-2	2.97E-2	–
23 × 23	9.53E-2	4.42E-2	–	3.52E-2	–	–
25 × 25	1.29E-1	5.67E-2	7.46E-2	5.07E-2	4.30E-2	8.29E-2
27 × 27	1.62E-1	6.95E-2	–	6.15E-2	–	–
29 × 29	2.12E-1	8.55E-2	9.90E-2	7.71E-2	5.97E-2	–
31 × 31	2.60E-1	1.03E-1	–	8.76E-2	–	–
33 × 33	3.26E-1	1.27E-1	1.27E-1	1.08E-1	7.43E-2	1.29E-1
35 × 35	3.85E-1	1.48E-1	–	1.28E-1	–	–
37 × 37	4.76E-1	1.76E-1	1.63E-1	1.54E-1	9.68E-2	–
39 × 39	5.55E-1	2.05E-1	–	1.86E-1	–	–
41 × 41	6.61E-1	2.39E-1	2.01E-1	2.13E-1	1.22E-1	2.25E-1
43 × 43	7.62E-1	2.73E-1	–	2.51E-1	–	–
45 × 45	8.94E-1	3.14E-1	2.50E-1	2.88E-1	1.60E-1	–
47 × 47	1.03	3.58E-1	–	3.30E-1	–	–
49 × 49	1.18	4.11E-1	2.97E-1	3.71E-1	1.90E-1	3.63E-1
51 × 51	1.31	4.55E-1	–	4.21E-1	–	–
53 × 53	1.49	5.15E-1	3.60E-1	4.80E-1	2.36E-1	–
55 × 55	1.67	5.75E-1	–	5.33E-1	–	–
57 × 57	1.86	6.43E-1	4.24E-1	5.98E-1	2.88E-1	4.57E-1
59 × 59	2.08	7.07E-1	–	6.67E-1	–	–
61 × 61	2.34	7.91E-1	5.00E-1	7.42E-1	3.56E-1	–
63 × 63	2.58	8.76E-1	–	8.31E-1	–	–
65 × 65	2.88	9.85E-1	5.77E-1	9.34E-1	4.02E-1	5.65E-1
67 × 67	3.10	1.06	–	9.94E-1	–	–
69 × 69	3.43	1.16	6.81E-1	1.10	4.75E-1	–
71 × 71	3.74	1.24	–	1.18	–	–

Table 3. (continuted) Experimental timing data (in seconds) for various grids and various machines. For the missing data in the table, "–" means that the grid is not compatible with the domain decomposition; blank spaces are for the data unavailable due to the memory limitation.

	iPSC/860			DELTA		
Grid Size	Sequential	4 processors	16 processors	4 processors	16 processors	64 processors
73 × 73	4.13	1.37	7.77E-1	1.30	5.80E-1	6.59E-1
75 × 75	4.47	1.47	–	1.40	–	–
77 × 77	4.87	1.62	8.81E-1	1.55	6.42E-1	–
79 × 79	5.25	1.74	–	1.67	–	–
81 × 81	5.73	1.90	9.95E-1	1.82	7.30E-1	7.85E-1
83 × 83	6.10	2.04	–	1.96	–	–
85 × 85	6.60	2.18	1.16	2.10	8.72E-1	–
87 × 87	7.11	2.31	–	2.24	–	–
89 × 89	7.66	2.52	1.30	2.40	1.03	9.62E-1
91 × 91	8.14	2.66	–	2.55	–	–
93 × 93	8.78	2.86	1.44	2.75	1.10	–
95 × 95	9.33	3.03	–	2.92	–	–
97 × 97	9.97	3.27	1.56	3.17	1.22	1.18
99 × 99	1.05E+1	3.44	–	3.32	–	–
101 × 101	1.13E+1	3.70	1.79	3.57	1.40	–
103 × 103	1.20E+1	3.90	–	3.82	–	–
105 × 105	1.25E+1	4.18	1.99	4.07	1.57	1.44