1991

Performance of PDE Sparse Solvers on Hypercubes

Mo Mu

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

Report Number:
91-002

PERFORMANCE OF PDE SPARSE SOLVERS ON HYPERCUBES

Mo Mu
John R. Rice

CSD-TR-91-002
January 1991

Performance of PDE Sparse Solvers on Hypercubes

Mo Mu* and John R. Rice†
Computer Science Department
Purdue University
Technical Report CSD-TR-91-002
CAPO Report CER-91-2

February 4, 1991

1. INTRODUCTION

This paper investigates various aspects of the performance of nonsymmetric sparse solvers for solving elliptic PDEs on distributed memory, message passing (DMMP) multiprocessors, typically, hypercubes. We use the Parallel ELLPACK system [Houstis, Rice and Papatheodorou, 1989] for this performance experiment. A complete PDE solver usually consists of five major components as illustrated in Figure 1: Domain Decomposition, Assignment, Discretization, Indexing and Solution. There is not a single optimal choice for any one of these components due to their mutual interactions and application requirements. A discussion of the algorithmic components one may choose to create a parallel PDE sparse solver is given in [Mu and Rice, 1990]. We do not intend to consider all the possible choices and combinations. We only briefly describe those of a few "good" combinations which are used in our performance experiment. We do show that the component choices have major effects on performance (which is no surprise) and that they interact in strong ways with each other and the hardware characteristics. Our thesis is that *there is probably no universally best choice for any of the algorithm components.* If this thesis is correct, it is a discouraging one as it implies that achieving very high performance requires the continuous creation of sparse matrix algorithms which exploit the special properties of the PDE problem and the hardware/software environment. Of course, we look forward to finding sparse solvers which provide good, even if not best, performance across a broad class of PDE problems and computing environments.

*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.

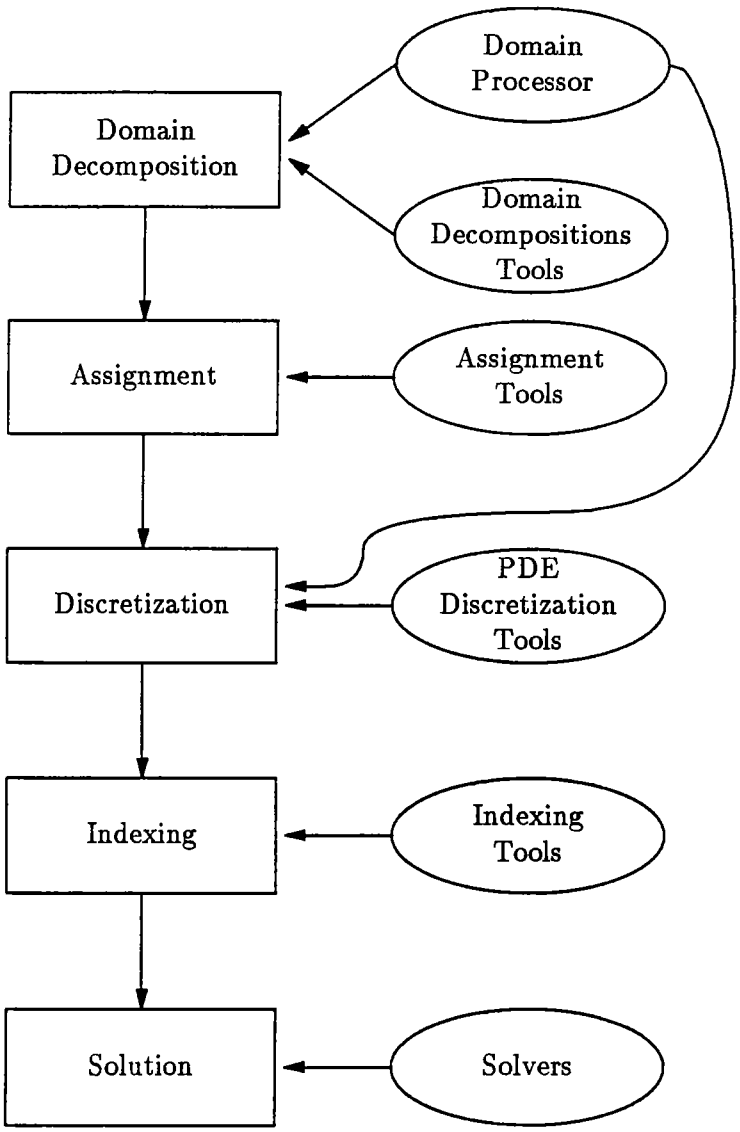
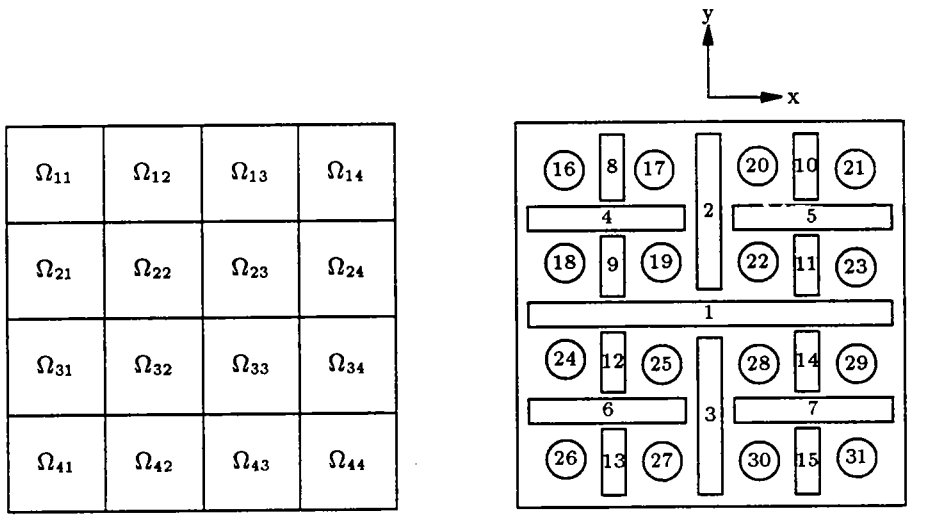


Figure 1: The five major components of a PDE solver within the Parallel ELLPACK structure. The sparse matrix solver must operate in this context.

1.1 Domain Decomposition

Domain decomposition consists of two phases. First, the geometric domain is decomposed into a set of **subdomains** by a square mesh. The number of subdomains is normally equal to the number of processors and we assume that is the case here. Second, the interface set, the grid lines of the square mesh, is partitioned into several levels suitable for a hypercube machine. Dissection in alternating directions is used, and each level consists of several **separators**. This is implemented as a domain decomposition module INCOMPLETE NESTED DISSECTION. An example of this decomposition is illustrated in Figure 2 for a rectangular domain. This approach can be extended to nonrectangular domains.



(a) Square mesh subdomains (b) Dissection interface separators

Figure 2: (a) The decomposition of a rectangular domain into 16 subdomains. (b) The interface sets (numbered 1 to 15) are structured into separator sets using an incomplete nested dissection method.

As explained in [Mu and Rice, 1990], this domain decomposition results in a hierarchic block independence of unknowns with respect to the subdomains and separators. This structure can be represented by a (block) elimination tree as shown in Figure 3. For simplicity, we refer to each tree node (a subdomain or separator) as a (generic) subdomain. Besides the obvious parallelisms

for a dense matrix, further parallelism can be exploited from the sparsity because independent tree nodes can be eliminated simultaneously during the Gauss elimination.

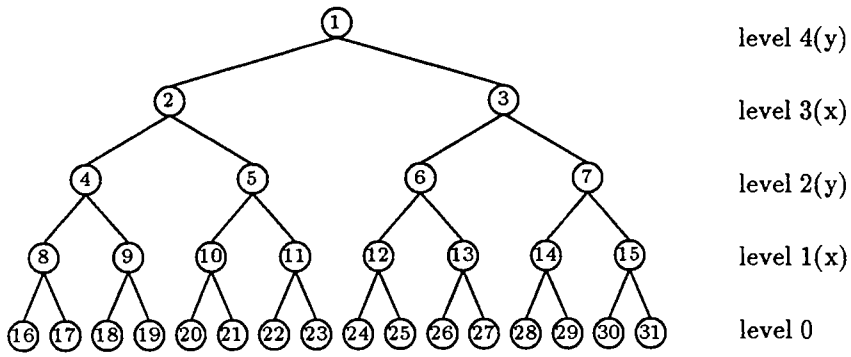


Figure 3: Block elimination tree produced by the incomplete nested dissection domain decomposition. The numbering of nodes corresponds to the groups of unknowns in Figure 2, the x, y levels refer to the directions of the bisection.

This incomplete nested dissection includes several variations depending on detailed structure of the interfaces. For example, amalgamating every two adjacent x and y levels gives the standard nested dissection. It is reasonable to amalgamate several, say, ℓ, top levels because the problem becomes denser and denser when the elimination moves from the bottom to the top of the elimination tree. If $\ell = L - 1$ where L is the total number of levels of the tree, then this becomes the capacitance matrix approach. In this case, no partition is used for the interface set. In the extreme case, i.e., $\ell = L$, this reduces to a complete dense solver, which makes no use of sparsity to exploit parallelism.

1.2 Discretization

Potentially, one can apply different discretization schemes to individual subdomains, adapting them to the local geometric and physical properties. For simplicity, we use the *5-point-star* scheme on a tensor product grid. As a matrix problem, there are several storage patterns. We choose the most natural storage by rows (equations).

1.3 Assignment

Two assignment methods are tested. One is the standard SUBTREE-SUBCUBE [George,

Table 1: Four types of local indexings that may be used within the separators.

Type	Description
0	wrapping in each separator (standard)
1	segmentwise-wrapping in all separators
2	segmentwise-wrapping in the top level separator only
3	segmentwise-wrapping in all separators but the top level

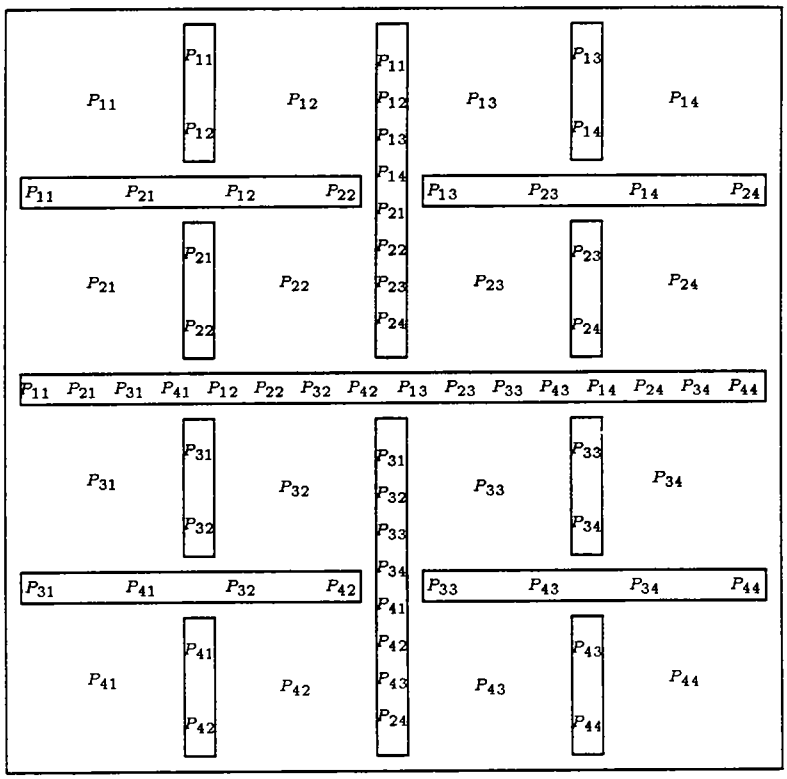
Liu, and Ng, 1987]. Here the root node of the elimination tree is first assigned to the whole hypercube and then the hypercube is split into two subcubes to which the two descendant subtrees are assigned. This process goes on recursively until one reaches single processors which are assigned the subdomain (leaves of the elimination tree). The assignment within each node is simply in the wrapping manner. The other method is the GRID BASED SUBTREE-SUBCUBE [Mu and Rice, 1991] which assigns each separator node based on the grid segments to the nearest processors in the subcube so that the communication front is the same as the elimination front. The assignment on the bottom level is in the mesh manner, i.e., the processors which are neighbors geometrically are also neighbors in the hypercube. These two assignments are illustrated in Figures 4 and 5 for 16 processors.

1.4 Indexing

We use a MODIFIED NESTED DISSECTION indexing module which is described as follows. Globally, the incomplete nested dissection ordering is used by blocks, compatible with the domain decomposition. For the local indexing within each subdomain Ω_{ij}, we order the interior unknowns on $\Omega_{ij} - B_{ij}$ first, followed by the unknowns on the boundary B_{ij} of Ω_{ij}. This uses the multifrontal idea to reduce communication by keeping the elimination fronts separate as long as possible. In principle, one can choose an arbitrary ordering for each $\Omega_{ij} - B_{ij}$, such as *rowwise left-to-right, nested dissection, minimum degree*, etc. For simplicity, we use the first one. For the local indexing within each separator, we use either the standard wrapping or the segmentwise-wrapping as illustrated in Figure 6. Table 1 lists four possible combinations of these two separator orderings.

1.5 Solution

The solution phase consists of factorization and substitutions. We only consider the performance of the factorization in this paper. Part of the reason is that there is, so far, no really efficient parallel



SUBCUBE-SUBTREE (**Standard**)

Figure 4: Standard subtree-subcube assignment for 16 processors. Within each box (separator) unknowns are assigned in wrapping manner to processors shown in the boxes.

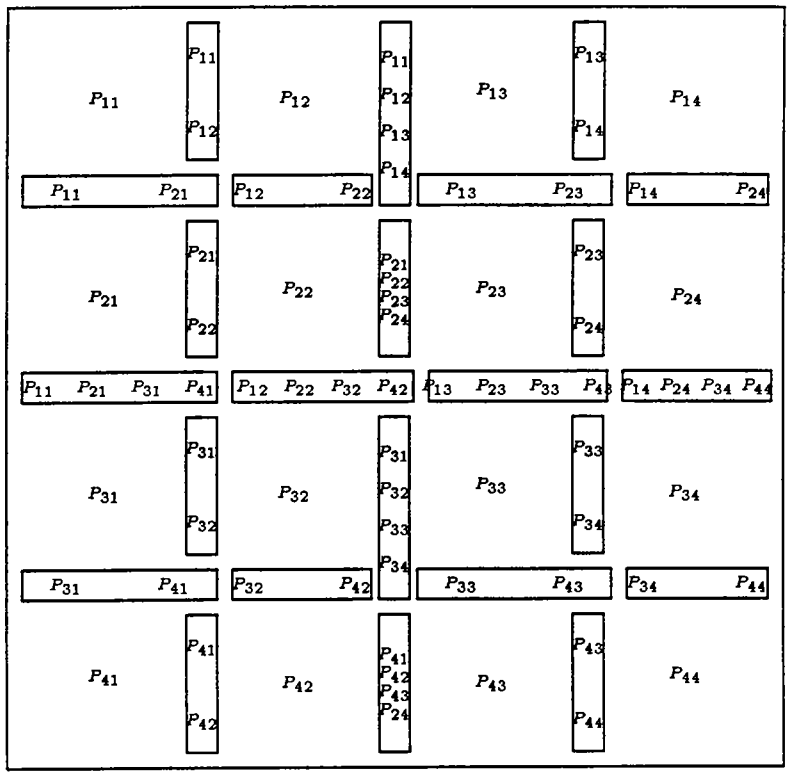


Figure 5: Grid based subtree-subcube assignment for 16 processors. Within the subdomain interfaces we show how the processors are assigned to unknowns in parts of the separators.

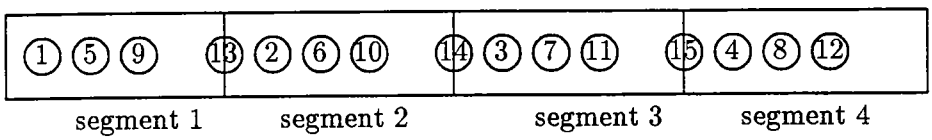


Figure 6: Segmentwise-wrapping in a typical separator.

algorithm for solving sparse triangular systems on DMMP machines.

Two solution modules are used to study various aspects of factorization algorithms. One is PARALLEL SPARSE which is basically a fan-out scheme. It is outlined as follows.

Algorithm 1: Distributed Sparse LU Factorization for a Processor P.

for level ℓ from bottom to top, do:

  for each node S, with equations assigned to P, on level ℓ, do:

    if $\ell = 0$, then

      elim_local(S)

    else

      elim_global(S)

      elim_local(S)

    endif

  end of S loop

end of ℓ loop

elim_local(S): P participates in eliminating unknowns in S by performing the associated modifications on equations assigned to P. For those equations of S assigned to P, it also calculates the corresponding multiplier vectors and sends them to other processors.

elim_global(S): P performs the modifications on its equations due to eliminating unknowns in the descendants of S which have no equations assigned to P.

The other is NEW GAUSS ELIMINATION [Mu and Rice, 1990] which is formulated as follows. Let the linear system from the PDE problem be written in its matrix form

$$A\mathbf{x} = \mathbf{f} \tag{1}$$

with

$$A = \begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \cdot & & \cdot \\ & & & \cdot & \cdot \\ & & & \cdot & \cdot \\ & & & A_p & B_p \\ C_1 & C_2 & \cdots & C_p & D \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{x}_p \\ \mathbf{x}_d \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{f}_p \\ \mathbf{f}_d \end{bmatrix} \tag{2}$$

This can further be reduced to

$$\begin{cases} U_i \mathbf{x}_i + \tilde{B}_i \mathbf{x}_d = \tilde{\mathbf{f}}_i, & i = 1, \ldots, p \\ \tilde{D} \mathbf{x}_d = \tilde{\mathbf{f}}_d \end{cases} \tag{3}$$

where

$$A_i = L_i U_i, \qquad i = 1, \ldots, p \tag{4}$$

are the standard LU factorizations and

$$\tilde{B}_i = L_i^{-1} B_i, \qquad \mathbf{f}_i = L_i^{-1} \mathbf{f}_i, \qquad i = 1, \ldots, p$$

$$\begin{cases} \tilde{D} = D - \sum_{i=1}^{p} C_i U_i^{-1} \tilde{B}_i \\ \tilde{\mathbf{f}}_d = \mathbf{f}_d - \sum_{i=1}^{p} C_i U_i^{-1} \tilde{\mathbf{f}}_i \end{cases} \tag{5}$$

Then, the NEW GAUSS ELIMINATION algorithm is outlined as follows.

Algorithm 2: A New Organization of Sparse Gauss Elimination.

Factorization.

- for $i = 1$ to p, do
  - computer L_i, U_i, $\tilde{B}_i$ by Gauss elimination on subdomain equations.
  - compute $\tilde{\tilde{B}}_i (= U_i^{-1} \tilde{B}_i)$ by back substitution

end i loop

- compute $\tilde{D}(= D - \sum_{i=1}^{p} \quad C_i\tilde{\tilde{B}}_i)$.
- compute L_d, $U_d(\tilde{D} = L_dU_d)$ by Gauss elimination on interface submatrix.

Solution.

- for $i = 1$ to p do
    - compute $\tilde{\mathbf{f}}_i$ and $\tilde{\tilde{\mathbf{f}}}_i(L_i\tilde{\mathbf{f}}_i = \mathbf{f}_i,\ U_i\tilde{\tilde{\mathbf{f}}}_i = \tilde{\mathbf{f}}_i)$ (forward/back substitutions).

end i loop

- compute $\tilde{\mathbf{f}}_d(= \mathbf{f}_d - \sum_{i=1}^{p} \quad C_i\tilde{\tilde{\mathbf{f}}}_i)$
- compute $\mathbf{x}_d(\tilde{D}\mathbf{x}_d = L_dU_d\mathbf{x}_d = \tilde{\mathbf{f}}_d)$ (forward/back substitutions).
- for $i = 1$ to p, do
    - compute $\tilde{\tilde{\mathbf{f}}}_i(= \tilde{\mathbf{f}}_i - \tilde{B}_i\mathbf{x}_d)$.
    - compute $\mathbf{x}_i(U_i\mathbf{x}_i = \tilde{\tilde{\mathbf{f}}}_i)$ by back substitution.

end of i loop

2. PERFORMANCE EXPERIMENTS

Our model problem is a *Poisson* equation with *Dirichlet* boundary condition on a rectangular domain. Even though the problem is actually symmetric, we still treat it as nonsymmetric since we want to examine the effects of algorithms for general problems.

Our approach is to experiment with most of the algorithm components fixed. We then vary a few components, often just one, observe the results and suggest conclusions about performance in general. We are also able to make comparisons with some performance data published by others involving particular algorithm components. Our experiments use the NCUBE 1 and NCUBE 2 machines, usually using 16 processors. We do no discuss their architecture here, see [Palmer, 1986], but note that they have considerably different performance parameters and yet both have the low communication to computation speed ratios typical of hypercubes and distributed memory machines in general. We list in Table 2 the module combinations for all experiments.

Table 2: Module Combinations used in the performance experiments.

Phase	Module		Experiment 1	2	3	4	5	6
Domain Decomposition	Incomplete Nested Dissection	mesh size	4×4	4×4	4×4	4×4	4×4	$2 \times 2; 4 \times 4$
		interface organization parameter ℓ	varying	1	1	1	1	1
Discretization	5-Point-Star		X	X	X	X	X	X
Assignment	Grid		X	X	X		X	X
	Standard			X				
Indexing	Modified Nested Dissection (ℓ)		$\ell = 2$	$\ell = 0$	ℓ is varied	$\ell = 2$	$\ell = 2$	$\ell = 2$
Solution	Parallel Sparse		X	X	X		X	
	New Gauss					X		X

Table 3: Factorization execution time in seconds using sparsity and parallelism to different extents within the block elimination tree.

Case $\ell =$	1	2	3	4	5
Factorization time	29.20	35.18	39.77	41.33	63.49

Experiment 1. Parallelism for various interface partitions.

This experiment uses a 4×4 square mesh for the domain decomposition, there are five levels in the elimination tree. We test how the amalgamation of the top ℓ levels decreases the efficiency of parallelism in using the sparsity represented in the elimination tree. Table 3 lists the factorization time on the NCUBE 1 with a 37×37 grid. On one hand, we see that the more sparsity is exploited, the more parallel efficiency is expected. Therefore, the incomplete nested dissection is better in parallelism than the standard nested dissection. On the other hand, the gain in efficiency decreases as ℓ decreases. Note that to utilize this sparsity parallelism, one must increase the formulation and implementation complexity. So, for larger problems, one might reach a point where further partition of the interface set would degrade the performance. In other words, there is trade-off between parallelism and actual performance in using sparsity during different stages of the Gauss elimination.

Experiment 2. Assignment effects.

Figure 7 shows that using different assignments has a big impact on the communication requirements, which also supports the theoretical analysis in [Mu and Rice, 1991]. Figure 8 shows the

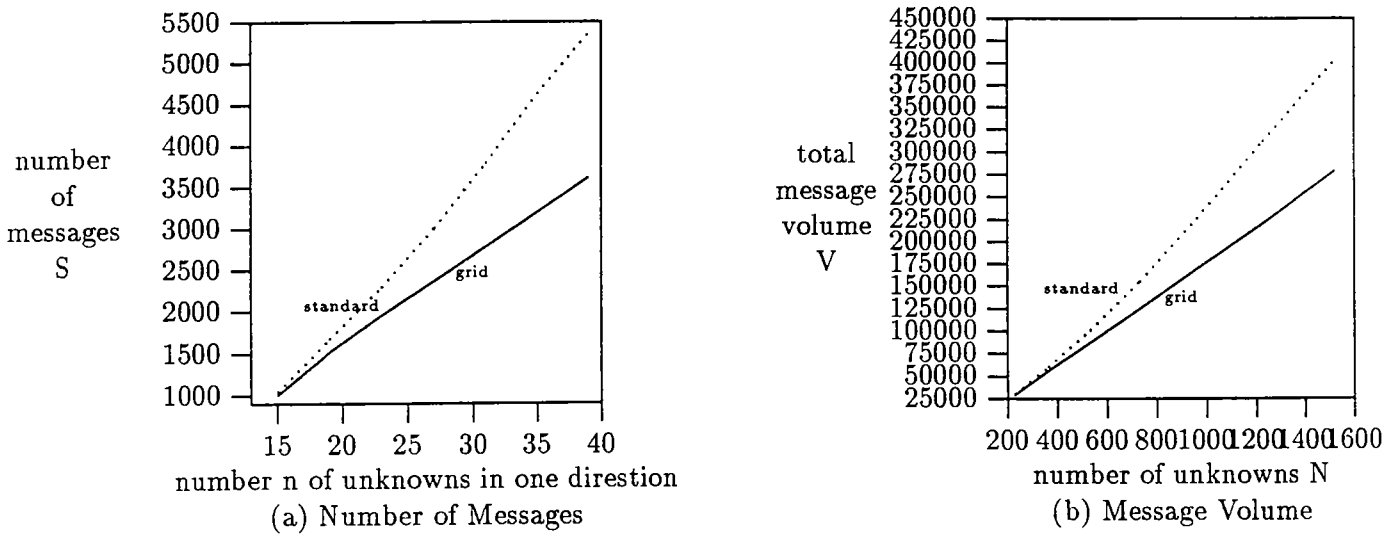


Figure 7: (a) Total number of messages S vs. number $n = N^{1/2}$ of unknowns in one direction. (b) Total message volume V vs. number N of unknowns. The advantage of the grid assignment over the standard assignment grows with the problem size.

individual timing curves of the 16 processors for the *grid* and *standard* assignments respectively. We see from these figures that a better assignment can have worse load balance if other components are not properly coupled with it. The reason for the load imbalance here is because the *grid* assignment uses the segment-wise strategy in each separator while the indexing uses the wrapping strategy. Therefore, the assignment for the local dense problem in each separator is handled algebracially in a block manner which is a well known cause of load imbalance. The algebraicly wrapping shown in Figure 8(b) provides well balanced execution as expected, but slower overall performance.

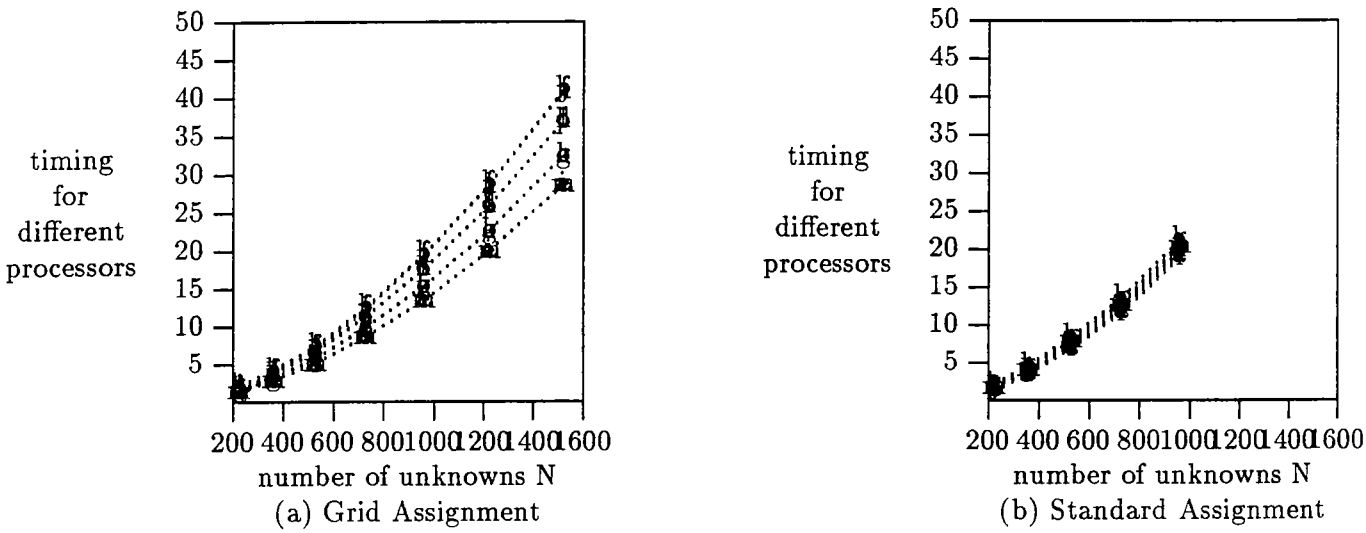


Figure 8: (a) Time vs. number N of unknowns for the *grid* assignment. The NCUBE 1 is used with 16 processors and the time is given for each processor (they naturally fall into four groups). (b) Time vs. number N of unknowns for the *standard* assignment. The NCUBE 1 is used with 16 processors and the time is given for each processor.

Experiment 3. Performance of indexing alternatives.

We next study how to vary the indexing to overcome the load imbalance and how this affects the overall performance. Table 4 includes the timing data for four indexing alternatives with a 37×37 grid. We observe that good load balance does not imply good performance. Indeed, the best performance is for the worst balanced and the worst performance is for the best balanced.

Table 4: The effects of 4 types of local indexings on the local balance and the overall timing performance. The minimum and maximum are over the 16 processors used.

Interface indexing Type	Maximum Time	Minimum Time
0	38.86	19.88
1	28.37	27.53
2	29.20	28.26
3	28.05	19.40

Experiment 4. Sparse matrix structure.

The key in devising efficient sparse algorithms is to properly utilize the sparsity for the given problem. Figure 9 shows the block structure of the original matrix. The sizes of these blocks change dramatically as indicated in Table 5. Figures 10–13 show how the sparsity varies during the elimination. We believe that using proper formulations, organizations, data structures and other techniques in different parts of the matrix and at different stages of the elimination is essential to achieving a high level of parallel performance.

Experiment 5. Effect of the lack of symmetry.

This experiment shows how the lack of symmetry seriously degrades the parallel performance of the sparse matrix Gauss elimination. We assume no symbolic factorization is used due to various reasons [see Mu and Rice, 1990]. Therefore, the algorithm of PARALLEL SPARSE has to monitor the column symbolic structure (C-INFO) which is used to generate the destination lists for communicating pivot equations (multicast tasks). This requires both manipulating the C-INFO data structure and extra communication. Our experiment is as follows. For the last (top) node of the elimination tree, we consider in **elim_local** that the multicast tasks are very close to broadcast tasks because the problem at this stage is almost dense. Therefore, the multicast is replaced by the broadcast and involving the C-INFO data structure is thus avoided. This approach cannot be used for all nodes since it introduces too much synchronization for sparse matrices as seen in Table 3. If one merely replaces the multicast by directly sending a message to all other processors (not a broadcast since no synchronization is used), then it will not only heavily increase the communication cost, but it also causes communication buffer overflow problems. Because our test problem is actually symmetric, we can, for all other tree nodes, make use of the information in the corresponding row instead of using C-INFO. This is what symmetric (Cholesky) sparse matrix

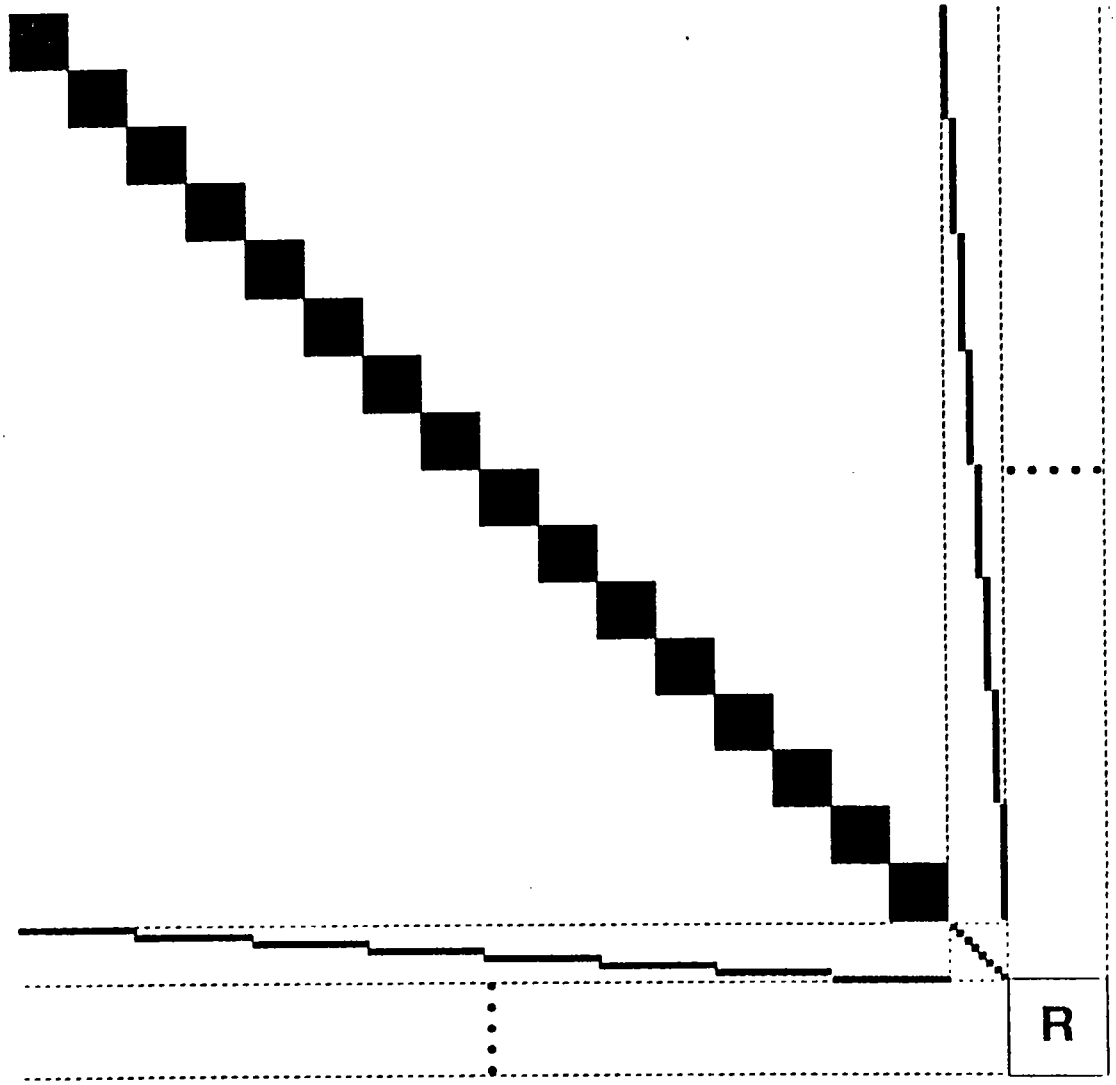


Figure 9: The sparse matrix structure for $p = 16$ processors. For the first two levels the solid boxes are where nonzero matrix elements might be (actually, these blocks are sparse also). The lower right box R contains diagonal blocks for the other 3 levels. Dots indicate sparse rows and columns. The relative sizes are correct for $n^2 = 100$, the number of grid points in one subdomain.

Table 5: Sizes of the diagonal blocks of the linear system for six cases. Given by level are: the number of diagonal subblocks, the total number of unknowns (equations) in this block, and the percentage of the total unknowns for this block. The level *Rest* is the total except levels 0 and 1. Here p = number of processors and n^2 = number of grid points in one subdomain.

	$p = 16$		$p = 64$		$p = 64$ (3 dimensions)	
Level	$n = 10$	$n = 30$	$n = 10$	$n = 30$	$n = 5$	$n = 10$
0: subblocks	16	16	64	64	64	64
order	1600	14,400	6400	57,600	8000	64,000
%	86.5	95.2	84.6	94.4	65.8	80.5
1: subblocks	8	8	32	32	32	32
order	80	240	320	960	800	3200
%	4.3	1.6	4.2	1.6	6.6	4.0
2: subblocks	4	4 16	16	16	16	
order	84	244	336	976	880	3360
%	4.5	1.6	4.4	1.6	7.2	4.2
3: subblocks	2	2	8	8	8	8
order	42	122	168	488	968	3528
%	2.3	0.8	2.2	0.8	8.0	4.4
4: subblocks	1	1	4	4	4	4
order	43	123	172	492	484	1764
%	2.3	0.8	2.3	0.8	4.0	2.2
5: subblocks			2	2	2	2
order			86	246	506	1806
%			1.1	0.4	4.2	2.3
6: subblocks			1	1	1	1
order			87	247	529	1849
%			1.1	0.4	4.3	2.3
Rest	1	1	1	1	1	1
order	169	489	849	2449	3367	12,307
%	9.1	3.2	11.2	4.0	27.7	15.5

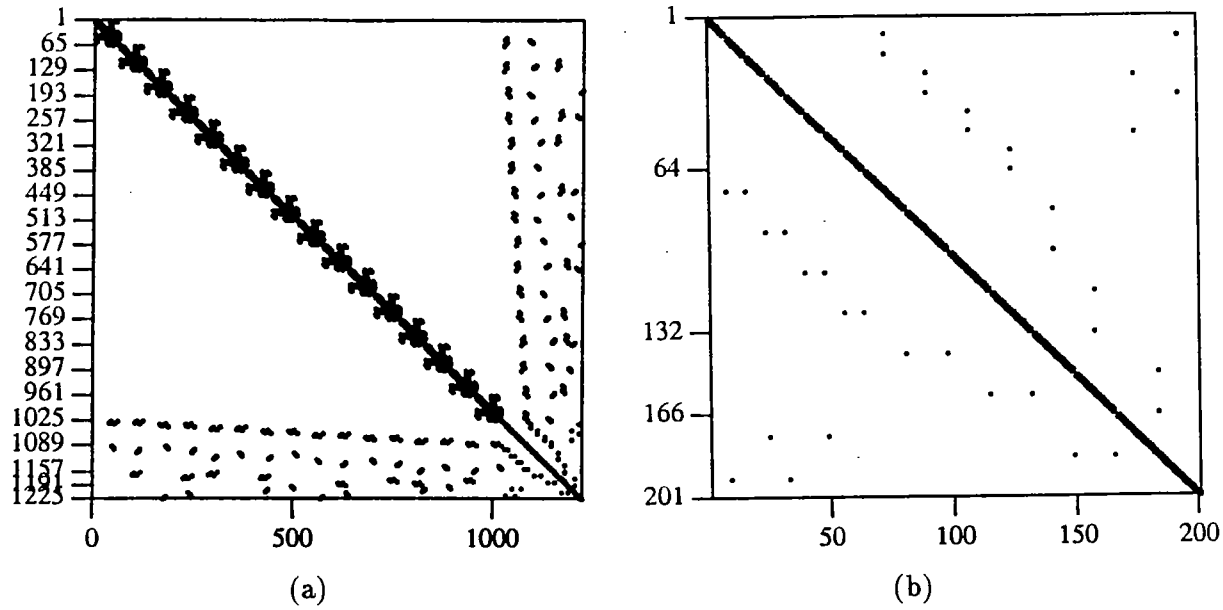


Figure 10: (a) Actual non-zero structure with $p = 16$, $n = 8$. The equation numbers are listed on the left. (b) The lower right block (everything except level 0) before the elimination starts.

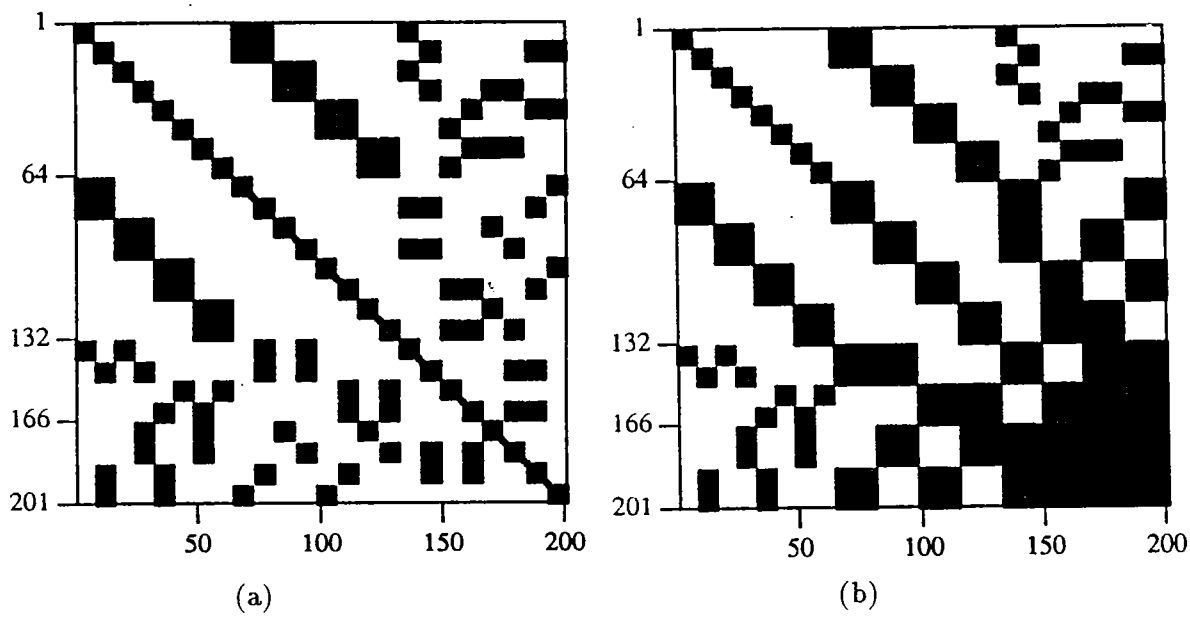


Figure 11: (a) The effect of the level 0 elimination on the lower right block. $\tilde{D}$ is given by (5). (b) The lower right block at the end of the elimination.

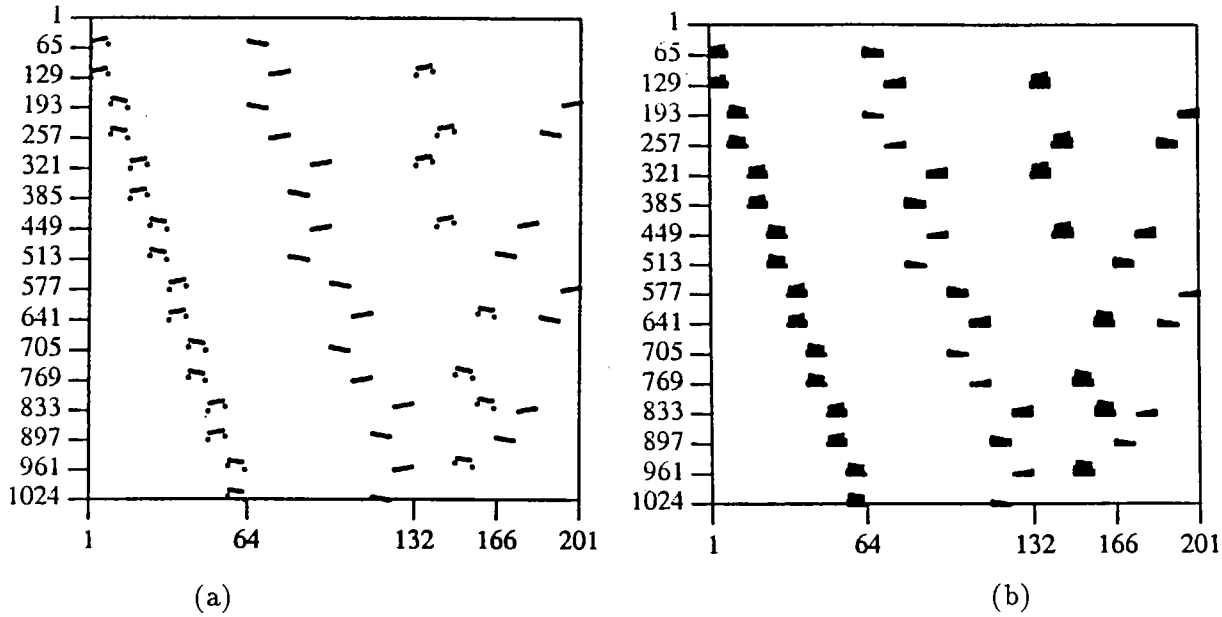


Figure 12: (a) The non-zero structure of the upper right matrix B before the elimination starts. Note that the display is distorted, B has 1024 rows and 201 columns. (b) The upper right matrix $\tilde{B}$ after the level 0 elimination.

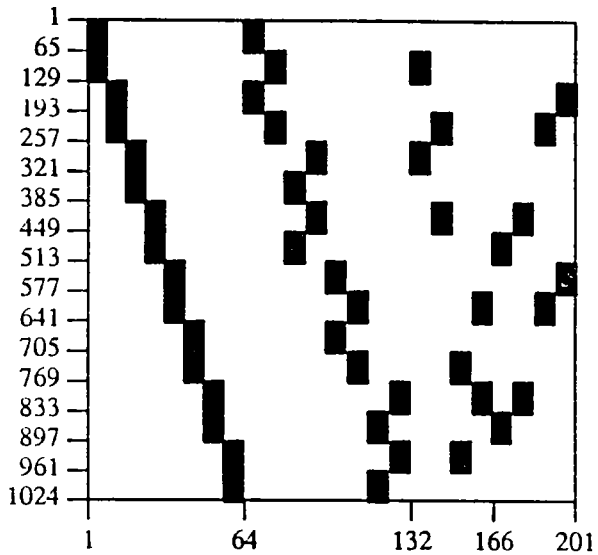


Figure 13: The final upper right matrix $\tilde{\tilde{B}} = U_{TOP}^{-1} L_{TOP}^{-1} B$ used to modify D before its factorization.

solvers do. This change makes the execution time drop dramatically from 29.20 seconds to 6.41 seconds.

In order to convert this data into speed up, we use the ELLPACK sequential module SPARSE GE to get the sequential timing of 39.93 seconds. This module is developed by A.H. Sherman from the zero-tracking code in the Yale Sparse Matrix Package. We see the very poor speed up 39.93/29.20 = 1.4 of PARALLEL SPARSE using 16 processors. Using symbolic factorization can also avoid processing the C-INFO, but the numerical factorization time cannot be improved to less than 6.41 seconds because the symbolic factorization usually creates more computations than the dynamic data structure does in PARALLEL SPARSE. Therefore, the speed up would be at most improved to 39.93/6.41 = 6.2 if the symbolic factorizations were included in PARALLEL SPARSE and assuming the extra preprocessing costs were not counted. All of the above timing data were obtained from the NCUBE 1.

Experiment 6. Performance of NEW GAUSS ELIMINATION

This new algorithm is devised by examing the above experiments. We make a few key points and refer to [Mu and Rice, 1990] for more details. First, we use different data structures to represent different parts of the matrix. For the subdomain equations, from Experiment 4 we see that they are very sparse, so the sparse data structure is used as usual. For the interface equations, however, we observe that only the lower right D part needs to be manipulated. This part becomes very dense after the elimination and is only a very small part in the whole matrix. Therefore, a dense matrix data structure is used to represent this part. For the algorithm organization, we know that the fan-in scheme is more efficient than fan-out in manipulating the sparse data structure, but fan-in is not applicable to solve nonsymmetric systems on a DMMP machine (see [Mu and Rice, 1991] for the reasons). Fortunately, for the level 0 subdomain equations, each processor holds the whole set of equations in its subdoman, so it is like a shared memory problem and the fan-in scheme can still be used at this stage. For the D part (which is distributed over all the processors), we apply the fan-out scheme. However, we do not have to manipulate the data structure because a dense matrix structure is used. The new organization also avoids using the column information, the (C-INFO) which degrades the performance so much as shown in Experiment 5. For communicating the subdomain equations, no column information is needed because the lower left part $(C_1, C_2, \ldots, C_p)$ is not changed in the new organization. For the interface system, we simply treat it as a dense matrix and make each pivot equation available to all processors. The speed ups of this algorithm are shown in Table 6. We see that it achieves almost full speed up. For comparison, we also present in Table 7 the speed ups reported in [Ashcraft, et. al., 1990] using 16 processors on an Intel machines for similar PDE problems. All the algorithms in Table 7 are for Cholesky factorization, so no nonsymmetric difficulties are present. We do not believe that the differences between the NCUBE and Intel machines account for a significant amount of the differences observed between Tables 6 and 7.

Table 6: Performance of the new Gauss elimination organization in factoring the matrix A for PDE problems. These speed ups were obtained on an NCUBE 2.

Decomposition	Processors	Grid	Unknowns	Speedup
2×2	4	23×23	441	4
2×2	4	33×33	961	4
4×4	16	37×37	1225	11.8
4×4	16	45×45	1849	13.6
4×4	16	61×61	3481	15.7

Table 7: Performance of other Cholesky PDE sparse matrix solvers reported by [Ashcraft, et al.] (no nonsymmetric difficulties present). These speed ups were obtained on an Intel bypercube using 16 processors.

Method	Problem	Unknowns	Speed up
fan-in	31×31 grid, nine-point-start	841	7.03
fan-in	63×63 grid, nine-point-start	3721	9.65
fan-in	125×125 grid, nine-point-start	7503	10.62
fan-out	2614 unknowns	2614	5.54
multifrontal	65×65 grid, nine-point-start	3969	9.5

3. BUFFER REQUIREMENTS

Various strategies can be used to communicate information within sparse matrix (and other) DMMP algorithms. We label the extremes as follows:

Write_as_early_as_possible. Whenever a value is computed which is to be used by another processor, send it off immediately. This way, no processor has to wait unnecessarily for data.

Write_as_late_as_possible. Do not send data until it is requested by another processor. This way, a processor's computations are not slowed down by unnecessary early communication.

Read_as_late_as_possible. Do not look for a value from another processor until it is actually needed. This way, a processor's computations are not slowed down by unnecessary early processing of communication buffers.

Read_as_early_as_possible. Frequently empty buffers of any values that have arrived from other processors. This way, a processor's communication buffers do not fill up and cause serious problems.

These extremes are called *write_early, write_late, read_late*, and *read_early*, respectively.

It is not obvious, or even easy to analyze, how these extremes and intermediate strategies affect the details of performance. However, the concept of *pipelining* has been introduced and used by many in recognition of the fact that write_early should be a good strategy to reduce synchronization delays, especially when communication is very expensive compared to computation.

Early attempts to use the superficially attractive combination of write_early and read_late led to many mysterious problems that were difficult to diagnose on the early, crude hypercube systems. When these problems were identified as communication buffer overflow, people changed to write_early, read_early strategies. This prevents overflow but it also clearly introduces an extra overhead (and programming complication) into the computation.

Several facts are relevant here

- The communication buffers are internal to the operating system. Their sizes are not easily available and change from time to time. Their sizes are not adjustable by the user and are a small part of the total memory of the machines. The operating systems tend not to use robust codes for processing them and mysterious results occur when they become full or nearly full.
- It is expensive to check buffers very frequently. Checking the buffer probably costs a fair number of machine operations, perhaps 5 to 15. Reading from a buffer costs something too but that has to be done eventually in any case. Checking the buffer after each equation is processed could easily double the cost of elimination in the early stages.
- In an "ideal" PDE application with excellent load balancing, almost all the values communicated are computed before any are needed elsewhere.
- The total number of values to be communicated is the same order of magnitude as the problem data. We roughly estimate the following sizes (in words) using an $N \times N$ grid on the domain:

PDE problem data:	$20\ N^2 \log N$
Level 0 communication:	$60\ N^2$
Top level communication:	$12\ N^2$

Thus, the potential need for buffer space approaches that of the memory for problem data.

From these observations we draw three conclusions:

1. The management of communication buffers is an intrinsic feature of sparse matrix solvers on DMMP machines. This management problem will not go away even if very high performance communication facilities are provided.

2. Simple management strategies can potentially increase execution times by a factor of two or reduce the memory available for the problem by a factor of two.

3. Better management strategies need some assistance from the hardware and/or operating system so that these buffers need be emptied only when they are nearly full.

4. REFERENCES

Ashcraft, C., S.C. Eisenstat and J.H. Liu (1990), "A Fan-in Algorithm for Distributed Sparse Numerical Factorization", *SIAM J. Sci. Stat. Comp.*, Vol. 11, pp. 593–599.

George, A., J. Liu, and E. Ng (1987), "Communication Reduction in Parallel Sparse Cholesky Factorization on a Hypercube", *Hypercube Multiprocessors* (M. Heath, Ed.), SIAM Publications, Philadelphia, PA, pp. 576–586.

Houstis, E., J. Rice, and T. Papatheodorou (1989), "Parallel ELLPACK", *Math. Comp. Simul.*, **31**, pp. 497–508.

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.

Mu, M. and J.R. Rice (1990), "The Structure of Parallel Sparse Matrix Algorithms for Solving Partial Differential Equations on Hypercubes", CSD-TR-976, Computer Science Department, Purdue University.

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

Palmer, J.R. (1986), "A VLSI Parallel Supercomputer", *Hypercube Multiprocessors*, (M.T. Heath, Ed.), SIAM Publications, Philadelphia, PA, pp 19–26.