Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports
Department of Computer Science

1991

An Organization of Sparse Gauss Elimination for Solving PDEs on Distributed Memory Machines

Mo Mu

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

Report Number:
91-080

Mu, Mo and Rice, John R., "An Organization of Sparse Gauss Elimination for Solving PDEs on Distributed Memory Machines" (1991). *Department of Computer Science Technical Reports.* Paper 919.
https://docs.lib.purdue.edu/cstech/919

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

AN ORGANIZATION OF SPARSE GAUSS ELIMINATION FOR SOLVINNG PDES ON DISTRIBUTED MEMORY MACHINE

Mo Mu
John R. Rice

CSD-TR-91-080
November 1991

AN ORGANIZATION OF SPARSE GAUSS ELIMINATION FOR SOLVING PDES ON DISTRIBUTED MEMORY MACHINES

Mo Mu*
and
John R. Rice†
Computer Sciences Department
Purdue University

November 7, 1991

Abstract

A computational arrangement of Gauss elimination is presented for solving sparse, nonsymmetric linear systems arising from partial differential equation (PDE) problems. It is particularly targeted for use on distributed memory message passing (DMMP) multiprocessor computers and it is presented and analyzed in this context. The objective of the algorithm is to exploit the sparsity (i.e., reducing computation, communication and memory requirements) and to optimize the data structure manipulation overhead. The algorithm is based on the nested dissection approach, which starts with a large set of very sparse, completely independent subsystems and progresses in stages to a single, nearly dense system at the last stage. The computational efforts of each stage are roughly equal (almost exactly equal for model problems), yet the data structures appropriate for the first and last stages are quite different. Thus we use different types of data structures and algorithm components at different stages of the solution. The new organization is a combination of previous techniques including nested dissection, implicit block factorization, domain decomposition, fan-in, fan-out, up-looking, down-looking, and dynamic data structures.

1 INTRODUCTION

Solving linear PDEs naturally generates large, sparse linear systems of equations to solve. These systems have structures which are not exploited by general purpose sparse matrix algorithms and we present a new computational organization of sparse Gauss elimination

*Supported by NSF grant CCR-8619817.

†Supported in part by AFOSR grant 88-0243 and the Strategic Defense Initiative through ARO contract DAAG03-86-K-0106.

tailored to exploit these structures and build an efficient parallel sparse PDE solver for Distributed Memory Message Passing (DMMP) multiprocessor machines. By sparse solver we mean that a direct method is used with some combination of sparse matrix techniques. Many PDE problems naturally generate non-symmetric matrices and we target these problems solved on DMMP machines, i.e., we pay particular attention to the minimization of communication costs of the algorithm. There has been considerable related work on parallel Cholesky factorization for symmetric positive definite matrices. We borrow some ideas from this work but there are fundamental differences between symmetric and non-symmetric algorithms for DMMP machines. One well known Cholesky algorithm is the fan-in scheme [1] which achieves a speedup of 10.62 using 16 processors on an Intel iPSC/2 with Weitek 1167 chips. This is for a PDE application using the nine point discretization on a L-shaped domain with a 125 by 63 grid. We start with some general background comments on sparse matrix methods for PDE problems.

The linear systems are almost always created by the PDE solving system with the equations (matrix rows) distributed among the processors. One has the freedom to choose the assignment of rows to processors, but one cannot choose to have columns assigned to processors without the high expense of performing a matrix transpose (or equivalent) on a DMMP machine. The linear systems are commonly non-symmetric so that a solver of symmetric systems is applicable to a limited class of PDE problems and/or discretization methods. For example, a Neumann condition applied on a curved boundary, or a collocation discretization generates a structurally non-symmetric matrix even with a Poisson equation. The lack of symmetry requires two data structures on a DMMP machine, one each for the row and column sparsities. One can combine these in clever ways, but it is prohibitively expensive to repeatedly obtain column sparsity information from the row sparsity structure. Note that similar sparsity patterns occur in other important applications (e.g., least squares problems [21]) and the considerations studied here for PDE problems are also relevant there. Pivoting is an important issue for direct methods which we ignore. Fortunately, for many discretizations of elliptic PDEs there are known indexings for the problem which are numerically stable. Finally, we note that iterative methods are often superior to direct methods for very large PDE problems. Unfortunately, the iterations do not always converge (or converge very slowly) so direct methods are also required.

Symbolic factorization is not well suited for non-symmetric systems as, so far, the techniques generate much too large a data structure. It is also not commonly used for PDE solvers since there is often only one right side of the linear system. Merging the symbolic factorization with the numerical computation is not inherently more expensive than doing these separately and, for non-symmetric systems, it allows the final data structure to be just the required size for the system. This dynamic data structure creation is used in the *parallel sparse* algorithm of [15].

If nested dissection is performed geometrically (as is natural in PDE problems) rather than algebraically then a great deal of matrix structure is known "a priori" from the geometric structure and need not be explicitly expressed in the sparse matrix data structure. This idea is exploited in *parallel sparse* [13] to some extent.

There are two general row oriented computational organizations of Gauss elimination focusing on how the LU factors are computed. They are the *up-looking* and *down-looking* organizations. Another important organization is the *multifrontal* scheme which is unknown oriented (using both rows and columns associated with unknowns). The up-looking organization processes the LU factorization row by row as follows.

Algorithm (*up-looking organization*)

- For $k = 1$ to n, do
  - form the k-th row of LU by modifying the k-th row of the original matrix using previously generated rows of LU.

  end k loop

Thus, at the k-th step, the unknowns x_i, $i = 1, 2, \ldots, k-1$ are eliminated from the k-th row, the first k rows of the L and U factors are completely known, and the remaining rows have not been touched. The alternative is the *down-looking* organization which processes LU by modifying the remaining submatrix using each pivot row.

Algorithm (*down-looking organization*)

- For $k = 1$ to $n-1$, do
  - use the k-th pivot row to modify the remaining rows

  end k loop

Here, in contrast with the up-looking organization, at the k-th step the unknown x_k is eliminated from the last $(n-k)$ rows to compute the k-th column of L and the $(k+1)$th row of U. Associated modifications are made to the remaining $(n-k)$ by $(n-k)$ block of the linear system. After the k-th step, the first k columns of L and the first $k+1$ rows of U have been completely computed and the remaining $(n-k-1) \times (n-k)$ matrix has been updated with the effects of the first k steps of the elimination.

Closely related to the algorithm organization are two communication organizations for DMMP machines. The *fan-out* organization has a processor send a row to a list of destination processors which need it to modify the rows they hold. The *fan-in* organization has a destination processor receive (from each contributing source processor) a partial sum which is an appropriate combination of rows held by the source processor. The partial sum, also called *modification vector*, is used to modify the current row held by the destination processor. There is an intermediate organization [11] which communicates a single row or partial sum using a control switch.

It is well known that the up-looking organization uses less data structure manipulation than the down-looking organization since up-looking does not process the sparse data structure for a row until it becomes the pivot row. Therefore, only one data structure

manipulation is required for each row by using a working buffer. The down-looking organization processes the data structure for a row several times, due to fill ins from all previous rows.

It is easy to see that fan-in organization requires less communication than fan-out. It is also easy to see that the up-looking and fan-in organizations naturally fit together, as do the down-looking and fan-out organizations. Thus the up-looking and fan-in is the preferred choice on the surface.

Unfortunately, there is a severe penalty for fan-in communication when the matrix is non-symmetric. Consider a source processor Q which holds

$$\{\text{row}_k,\ k \in K = \text{some set of indices}\}$$

and a destination processor P which holds row_i. The partial sum which Q is to send to P is

$$\sum_{k \in K} m_{ik} * \text{row}_k \quad \text{where} \quad m_{ik} = a_{ik}/a_{kk}$$

However, the multipliers m_{ik} (the a_{ik}) are in processor P (in row_i) and Q can obtain them only if P sends them. Further, the a_{ik} are not known until the eliminations are complete in row_i for all a_{ij}, $j < k$. This may involve many other processors. Note that for symmetric matrices $a_{ik} = a_{ki}$ and the multipliers are known to processor Q. It appears there is no way to use fan-in communication for non-symmetric problems without the penalty of communicating these multipliers. This penalty negates all the advantage of fan-in. It has been proposed [6] to have each processor hold rows and columns in pairs to overcome this difficulty. This can be effective for some matrix problems, but for PDE applications it requires a substantial data redistribution at the start as well as doubling the storage.

A disadvantage of the fan-out organization is that one must have large communication buffers to hold information sent out to be used later on other equations. For shared memory machines this information need not be actually moved (one can create pointers to it) but on DMMP machines this is a serious problem. Many early sparse matrix codes on these machines failed mysteriously due to communication buffer overflow (messages were received at a processor and accumulated unread). This overflow can even be a problem with the fan-in organization and is particularly frustrating because these buffers are not under programmer control. The current technology seems to provide no easy solution, one sees codes with frequent seemingly unnecessary read statements just to keep these buffers from overflowing. Another disadvantage of the fan-out is that it favors the inefficient down-looking computation organization. To fit the fan-out with the up-looking, each processor needs a big storage buffer to hold information received for the current working row and to be reused later on other equations.

We must create a computational organization which overcomes or minimizes the above difficulties in order to build an efficient parallel PDE solver for DMMP machines. It is already observed in [16] that there are several components to a sparse matrix solver and there is not an optimal choice for any one of them due to their mutual interactions and the

effect of application properties. Our experience suggests that, for PDE applications, the nested dissection ordering should be used in one way or another. If this is done we observe that the nature of the linear subsystems solved changes completely during the stages of solution and thus no one set of sparse matrix components can provide good efficiency throughout the stages of nested dissection. Since the nature of nested dissections is to equidistribute the work among the stages, inefficiency at any stage implies inefficiency of the entire computation.

Our task is to combine ideas for data structures, arithmetic algorithms and communication organizations so as to build an efficient solver. We use two ideas previously used in *parallel sparse*, namely, geometric information and dynamic data structures. Our algorithm has only two phases: solution of the first set of subsystems in the nested dissection, i.e., the subdomain equations corresponding to the geometric domain decomposition, and solution of the interface equations. We argue that our solution of the interface equations is efficient enough to get close to optimal efficiency. However, it is also clear that the interface equations have structure that we do not exploit and there may well be applications or machines where the gain in efficiency here is sufficient to warrant developing a more complex algorithm.

The remainder of the paper is organized as follows. Section 2 briefly describes a model problem to set the context of our algorithm. Section 3 presents the computational organization of sparse Gauss elimination and the modifications for a distributed parallel version are given. Section 4 reports on the algorithm's performance. The final section has brief comments on performance, extensions and conclusions.

2 THE APPLICATION CONTEXT

The application is described in [13] and [16]. Thus we are quite brief here. The model linear PDE problem is discretized on a rectangular domain which is divided into $p = 2^{2m}$ subdomains by nested dissection as indicated in Figure 1 for the case $m = 2$. We have p processors in the computation (16 for Figure 1) and n^2 unknowns in each subdomain. There are also unknowns in the separators that partition the domain. In general there are $(2^m n + 2^m - 1)^2$ unknowns in the problem.

The idea is to order the equations so as to solve the p subsystems on the subdomains, then solve the subsystems on the separators level by level in the order smallest to largest. at each stage of this process one has a set of completely independent subsystems (a factor of 2 fewer at each stage) which can be solved in parallel. The number of processors applied to each subsystem increases by a factor of two at each stage.

The matrix structure is illustrated in Figure 2 for $p = 16$ where the details are given only for the first two stages. The structure in the box labeled R (for rest) follows the same pattern, there are block diagonal matrices with 4, 2 and 1 blocks within this box. The final block is essentially dense when the computation reaches it and is of the order $4n + 3$ (in general its order is $2^m n + 2^m - 1$). The upper right set of columns and lower left set of rows (where the dots are) have a sparse structure not illustrated here. For efficient computation

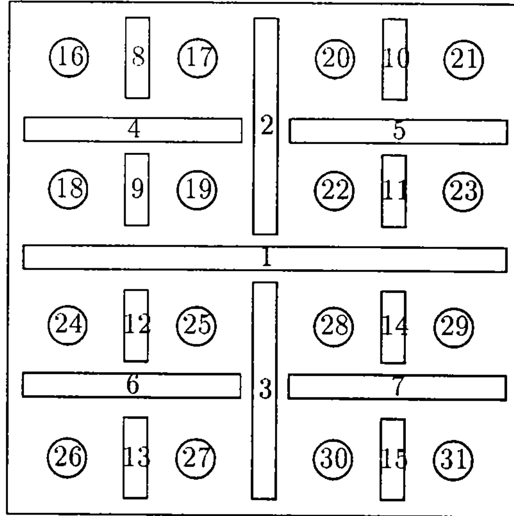

Figure 1: The geometric partition of a rectangle into p subdomains (for $p = 16$) by nested dissection. There are n^2 unknowns in each subdomain and a "line" of unknowns in each separator used to partition the rectangle.

the value of n must be large enough to give considerable work for each processor at the first level. For 16 processors, $n = 10$ is on the small side (100 equations per processor) and $n = 30$ is perhaps more appropriate (900 equations per processor). The sizes of these blocks change dramatically as indicated in Table 1. It gives the sizes of the first block for four cases: $p = 16$ with $n = 10$ and 30; and $p = 64$ with $n = 10$ and 30. An analogous approach for the cube in three dimensions in possible and we give the data for $p = 64$ with $n = 5$ and 10. Percentages of the total size are also given. Note that the diagonal subblocks of the diagonal blocks are themselves very sparse matrices at the lower levels. Two facts are clear from these examples:

1. The first group of unknowns, level 0, comprises the bulk of the sparse matrix. Recall, however, that the work to solve the systems on each level is roughly equal due to the further sparsity not displayed in Figure 2.

2. The balance of sizes for three dimensional problems is quite different than for two dimensional problems. Thus strategies which are quite efficient in two dimension might not be so in three dimensions.

Finally, we note the importance of exploiting the finer sparse structure not shown in Figure 2. For the case $p = 16$ and $n = 30$, the work to solve the 16 level 0 subsystems treating them as dense matrices is about $16 \times (900)^2 = 13/3 = 4$ billion arithmetic operations compared to about $16 \times (900)^2 = 13$ million operations using a band matrix method and about $16 \times 900 \times 5 \times 10 = 720$ thousand operations using nested dissection. Similarly, treating the "Rest" equations as a dense matrix problem requires about 40 million operations

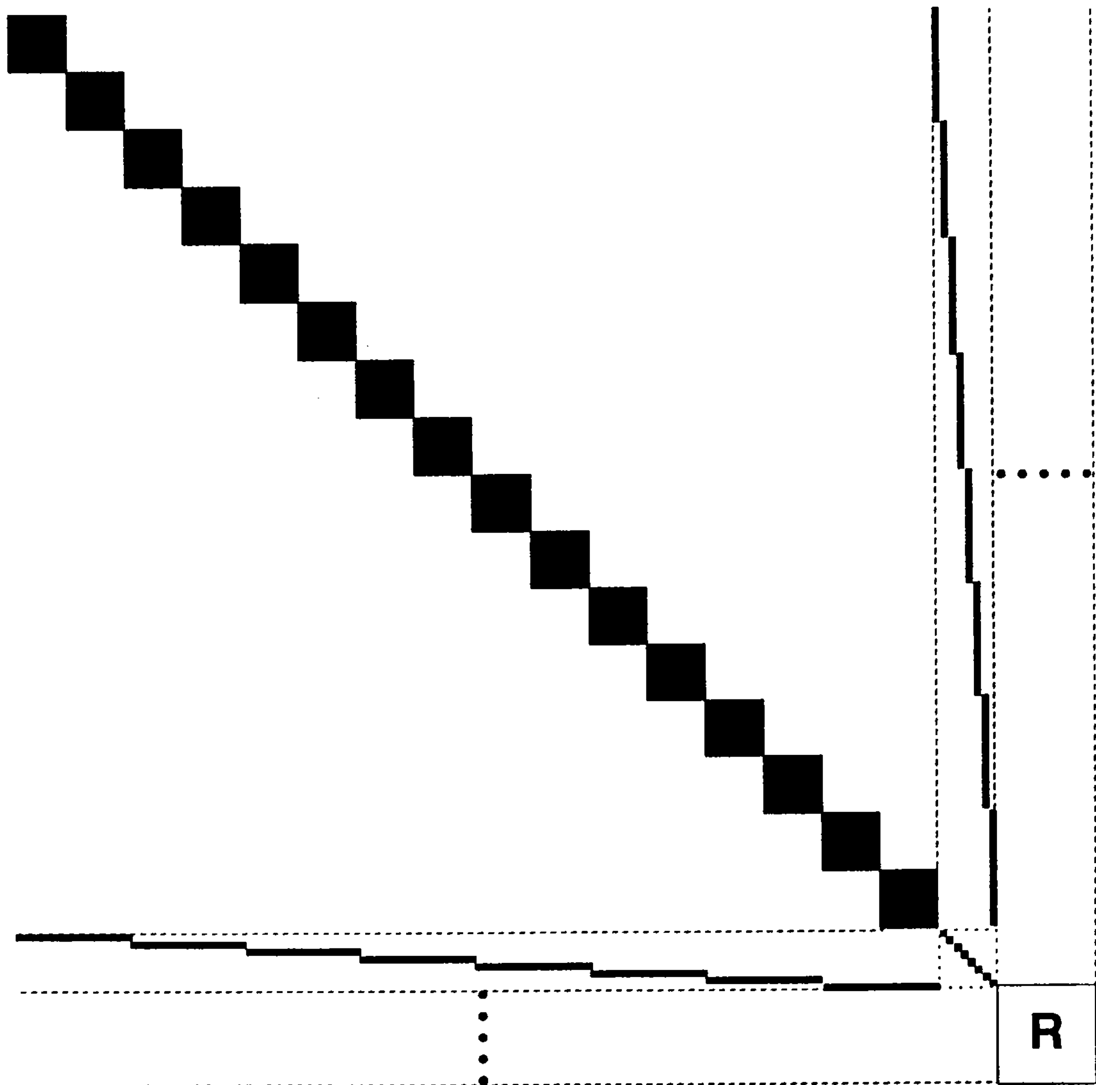

Figure 2: The sparse matrix structure for $p = 16$. For the first two levels the solid boxes are where nonzero matrix elements might be (actually, these blocks are sparse also). The lower right box labeled R (for Rest) is where the diagonal blocks for the other 3 levels are located. There are sparse rows (lower left) and columns (lower right) where the dots are. The relative sizes are correct for $n = 10$

Table 1: Sizes of the diagonal blocks of the linear system for six cases of practical interest. For each level we give, in order, the number of diagonal subblocks, the total number of unknowns (equations) for this diagonal block and the percentage of the total unknowns for this diagonal block. The level *Rest* is the total except levels 0 and 1 (see Figure 2).

Level	$p = 16$		$p = 64$		$p = 64$ (3 dimensions)	
	$n = 10$	$n = 30$	$n = 10$	$n = 30$	$n = 5$	$n = 10$
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

compared to about $3 \times (123)^3/3 = 1.9$ million using nested dissection on levels 2, 3 and 4. Similar numbers for the three dimensional problem with $p = 64$, $n = 10$ are, for level 0: 21 billion, 64 million and 3.2 million; for the "Rest" (levels 2–6): 6.2×10^{11} and $10^{10} = 10,000$ million. Note that the large size of this last number strongly suggests that there is a better way to do nested dissection in three dimensions than envisaged here.

3 THE COMPUTATIONAL ORGANIZATION

The linear system from the PDE problem can be written in its matrix form

$$M\mathbf{x} = \mathbf{f} \tag{3.1a}$$

with

$$M = \begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_p & B_p \\ C_1 & C_2 & \dots & C_p & D \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \\ \mathbf{x}_d \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \\ \mathbf{f}_d \end{bmatrix} \tag{3.1b}$$

The matrices C_i and D contain all the elements of levels 1 to 2^m described in Section 2 (see Figure 2). It is well known that the interface unknown vector $\mathbf{x}_d$ satisfies the following interface subsystem

$$S\mathbf{x}_d = \mathbf{g}_d \tag{3.2.a}$$

where

$$S = D - \sum_{i=1}^{p} C_i A_i^{-1} B_i \tag{3.2.b}$$

is often called the *Schur complement* or the *capacitance matrix*, and

$$\mathbf{g}_d = \mathbf{f}_d - \sum_{i=1}^{p} C_i A_i^{-1} \mathbf{f}_i. \tag{3.2.c}$$

It is generally believed that explicitly computing S in direct methods is too expensive. Iterative methods solve both the interface subsystem (3.2.a) and the subdomain subsystems

$$A_i\mathbf{x}_i = \mathbf{f}_i - B_i\mathbf{x}_d \tag{3.3}$$

iteratively, while the hybrid methods solve (3.2.a) iteratively and (3.3) directly. These methods are the common practice in the currently active domain decomposition and substructuring approaches. However, there are many cases where iterative methods do not converge or are too slow (many preconditioners only work well for model problems) and direct methods have to be used.

We can also relate this two level subdomain – interface approach to *implicit block factorization* [4] by interpreting the following relation

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} A & 0 \\ C & I \end{bmatrix} \cdot \begin{bmatrix} I & \hat{B} \\ 0 & S \end{bmatrix} \tag{3.4}$$

with $A\hat{B} = B$ and $S = D - C\hat{B}$. By *implicit* we mean that *only the factors of A and S are stored, but $\hat{B}$ is not, and when $\hat{B}$ is needed $A^{-1}B$ is used instead.* This basic idea can even be traced back earlier to, for example, [2] and [8], where it was used to save storage and to reduce computation in the backsolve. Little attention has been given in the literature to efficiently computing S by exploiting the sparsity available, though [8] gave two algorithms, one of the form

$$S_{*k} = D_{*k} - (CU^{-1})(L^{-1}B_{*k}), \quad \text{for all } k \tag{3.5}$$

and the other of the form

$$S_{*k} = D_{*k} - C(U^{-1}(L^{-1}B_{*k})), \quad \text{for all } k \tag{3.6}$$

where $A = LU$.

3.1 The Sequential Version

We now present a sequential algorithm which is efficient in computing S for DMMP machines. The basic idea is still to use (3.2.b) in the form

$$S = D - \sum_k C_{*k}(U^{-1}(L^{-1}B))_{k*}. \tag{3.7}$$

The matrix $\tilde{B} = L^{-1}B$ is computed as in the standard Gauss elimination on the first set of rows $(A\ B)$, it is stored by updating B. Then we note that a row of $(U^{-1}\tilde{B})_{k*}$ is not actually needed if the corresponding column C_{*k} is null. Thus we only need to explicitly compute a small number of the rows of $\hat{B}$ if C has mostly null columns. This is the case in our application where we will deal with $C_i\hat{B}_i = C_iA_i^{-1}B_i$ for each subdomain i and only the subdomain boundary layer unknowns have non-null columns in C_i. The discretization only connects these unknowns to the interface set. This fact has a physical explanation as follows. One can view $\hat{B}_i$ as the discrete Green's function. The process of reducing the global problem to the interface problem is equivalent to mapping the boundary condition on the interface for each subdomain from Dirichlet to Neumann. The derivative in the Neumann condition on the subdomain boundary is calculated using the Green's function to obtain values on the adjacent boundary layer from the Dirichlet data. Therefore, the Green's function needs to be evaluated only on that layer instead of on the whole subdomain.

To compute the k-th row $\mathbf{r}_k^T = (U^{-1}\tilde{B})_{k*}$, we can simply use

$$U^T\mathbf{y}_k = \mathbf{e}_k \tag{3.8}$$

$$\mathbf{r}_k^T = \mathbf{y}_k^T\tilde{B} \tag{3.9}$$

with the vector $\mathbf{e}_k^T = (0, 0, 1, 0, \ldots 0)$, with 1 as the k-th component. Note that the first $(k-1)$ elements of $\mathbf{y}_k$ are zeros, therefore, only the remaining smaller lower triangular system needs to be solved in (3.8) and computing (3.9). Thus computing the terms $C_{*k} \cdot \mathbf{r}_k^T$ is still a sparse matrix computation.

When C has a large number of null columns, this way of computing S has very a moderate extra computation compared with using (3.5) which is ordinary Gauss elimination. It is much cheaper than (3.6) which only uses B's sparsity when some of its columns are null. In addition, the extra arithmetic cost of (3.8) is compensated by avoiding the manipulation of the data structure of C. In [23] solving nonlinear systems of block bordered circuit equations using Newton's method is considered. Their Jacobian is sparse with a structure similar to that of Figure 2 and they also apply the block factorization using the explicit $\hat{B}$ throughout.

We have tested this variation on grids of size up to 61×61 and its speed is almost the same as using ordinary Gauss elimination for the LU factorization. However, it is very beneficial for parallel computation as seen in Section 3.2.

We now give a complete description of this sequential algorithm and call it *Modified Implicit Block Factorization.*

Algorithm: Modified Implicit Block Factorization

Factorization.

- Step 1 Perform Gauss elimination on subdomain equations.

  for $i = 1$ to p, **do**

  compute and store L_i, U_i, $\tilde{B}_i$ by

$$A_i = L_i U_i \tag{3.10}$$

$$L_i \tilde{B}_i = B_i \tag{3.11}$$

  end i loop

- Step 2 Compute Schur complement.

  for $i = 1$ to p, **do**

  for $((C_i)_{*k} \neq \text{null})$ **do**

  solve for $\mathbf{y}_k$ from

$$U_i^T \mathbf{y}_k = \mathbf{e}_k \quad (\text{matrix order of } n^2 - k + 1) \tag{3.12}$$

  computer $\mathbf{r}_k^T$ by

$$\mathbf{r}_k^T = \mathbf{y}_k^T \tilde{B}_i \tag{3.13}$$

  modify D by

$$D := D - (C_i)_{*k} \mathbf{r}_k^T \tag{3.14}$$

end k loop
end i loop ($S = D$)

- Step 3 Factor Schur complement.
  compute L_s, U_s and store them in D by

$$S = L_s U_s \tag{3.15}$$

Solution.

- **for** $i = 1$ to p **do**
  solve for $\tilde{\mathbf{f}}_i$ and store in $\mathbf{f}_i$ from

$$L\tilde{\mathbf{f}}_i = \mathbf{f}_i \tag{3.16}$$

end i loop.

- compute $\mathbf{g}_d$ and store in $\mathbf{f}_d$ by

$$\mathbf{g}_d = \mathbf{f}_d - \sum_{i=1}^{p} C_i(U_i^{-1}\tilde{\mathbf{f}}_i) \tag{3.17}$$

- solve for $\mathbf{x}_d$ from

$$L_s U_s \mathbf{x}_d = \mathbf{g}_d \tag{3.18}$$

- **for** $i = 1$ to p, **do**
  solve for $\mathbf{x}_i$ from

$$U_i \mathbf{x}_i = \tilde{\mathbf{f}}_i - \tilde{B}_i \mathbf{x}_d \tag{3.19}$$

end i loop.

3.2 The Distributed Parallel Version

To avoid repeating the algorithm details, we only make some comments about how to modify the above algorithm for a distributed, parallel computing environment.

1. Assignment

For p subdomains of the PDE application we have the subdomain equations

$$A_i \mathbf{x}_i + B_i \mathbf{x}_d = \mathbf{f}_i$$

stored in the ith processor for $i = 1, 2, \ldots, p$. The interface equations

$$\sum_{i=1}^{p} C_i \mathbf{x}_i + D\mathbf{x}_d = \mathbf{f}_d$$

are assumed to be assigned to processors equation by equation in some manner, for example, the subtree-subcube assignment [14], [9].

2. Data Structures

For the L_i U_i and $\tilde{B}_i$ of the subdomain equations, we can use the standard row-wise compressed sparse data structure [17]. One can see from the algorithm in Section 3.1 that it is more suitable to use a column oriented compressed data structure for the C part which is not changed during execution. We use a dense matrix data structure for the distributed interface submatrix D (and its final version the Schur complement S) by allocating a standard two dimensional array. This part of the storage requirements is comparably small and down-looking is much more costly using a sparse data structure.

3. Solving the Subdomain Equations

The first step of the algorithm can be totally parallelized without any communication since it only involves local computations on each processor. The efficient up-looking organization can be applied to (3.10) and (3.11).

4. Computing the Schur Complement

Step 2 is divided into two phases by observing that operations in (3.14) can be performed in any order with respect to k. In Phase 1 each processor computes the $\mathbf{r}_k$'s from (3.12) and (3.13) and sends them to other processors. This is also totally parallelized but with communication involved. This is the major phase where communication buffer overflow might occur if processors do not attempt to clear the buffers by receiving (reading) messages. However, the communication organization here is fan-in. Each $\mathbf{r}_k$ can be viewed as a partial sum (or *modification vector* in the fan-in terminology) with $\mathbf{y}_k$ as multipliers. This reduces the number of messages (or equivalently, communication start ups) which is a major part of communication cost on many DMMP machines. More importantly, the message volume is also much less than in the ordinary fan-in organization since only the interface part of $\hat{B}_i$ needs to be communicated instead of all of $(U_i,\ \tilde{B}_i)$. For example, we have run problems of up to a 100×100 grid without buffer overflow whereas using the standard fan-out organization we experienced buffer overflow crashes starting at a 41×41 grid.

The second phase in Step 2 is to perform (3.14). Synchronization delay is possible if a $\mathbf{r}_k$ has not arrived yet from another processor. This raises the question of how to choose the best order in Phase 1 to reduce the synchronization delays in Phase 2. This question is also related to the load balance problem, see [19] for details. In this paper we assume that k is in increasing order in the k loop. The computations (3.14) for updating D which correspond to the local $\mathbf{r}_k$'s can also be moved from Phase 2 to Phase 1 and performed right after each $\mathbf{r}_k$ has been computed. Whether $\mathbf{r}_k$ is used in (3.14), immediately or temporarily stored until Phase 2 depends on whether one is using the pipelining technique (see [19] for definition) or not. The algorithm and performance data presented here has $\mathbf{r}_k$ sent out immediately, then used in (3.14) and then discarded. Another important property of this algorithm is that in Step 2 the destination processors of the messages are statically determined by C_i's structure instead of the $(C_iU_i^{-1})$ structure as in the ordinary factorization. Thus the

message destination lists can be determined once and for all at the start rather than being generated dynamically and requiring repeated manipulation.

5. Factoring the Schur Complement

Step 3 has the standard parallelism implied by the elimination tree due to the nested dissection ordering used for the interface separators. Synchronization between subcubes is required and subcube computations gradually merge into the whole hypercube, level by level. Fan-out communication and down-looking computation organizations are used because of the nonsymmetry of the matrix. However, the down-looking organization is no longer inefficient because a dense matrix data structure is used for the Schur complement and thus avoiding the manipulation of sparse data structures. Note that using a dense data structure by no means implies using a dense matrix factorization algorithm. The computations here still exploit the sparsity available from the nested dissection. In addition, we simply put all processors on the destination lists since the matrices now become smaller and smaller, denser and denser, and little is gained by maintaining the destination lists. The implementation details in this part are very similar to a fan-out, nonsymmetric parallel sparse solver in [13], [15]. We note that similar ideas in Step 1 and Step 2 can be more or less applied recursively to Step 3 due to the nested dissection organization, this will be investigated in the future and will be of particular interest in the case of PDE in three dimensions.

4 PERFORMANCE

The above discussions show how the algorithm presented here addresses all the major difficulties in building a distributed nonsymmetric PDE sparse solver. This algorithm has been implemented and tested on the NCUBE/2, the Intel iPSC/2, and the Intel iPSC/860 and is installed in the parallel ELLPACK system [10]. To compute speed up we use the sequential sparse Gauss elimination module in ELLPACK [22] running on one node of these hypercube machines. This module is a version of the zero-tracking code in the Yale Sparse Matrix Package [6] written by A. Sherman.

Our test problem is to solve a model Laplace equation with Dirichlet boundary conditions on a rectangle. The five-point star discretization is used with tensor product grids. We treat the generated systems as nonsymmetric ones algorithmically though they are actually symmetric. Using 16 processors with a uniform 4×4 domain decomposition, we show in Figure 3 the growth of speed up as the grid size grows up to a 65×65 grid on the NCUBE/2. There is no obvious improvement of speed up as the grid size further grows. The corresponding timing data are listed in Table 2.

The algorithm performance is further improved when one incorporates with some other optimization strategies, such as unstructured scheduling and message packing [19]. For example, the parallel time is reduced to 1.54 seconds on the 57×57 grid and to 1.87 seconds on the 61×61 grid. The speed up is thus improved to 10.89 and 11.15, respectively.

We have the performance similar to Table 2 for the Intel iPSC/2. However, when

Table 2: Timing results (in seconds) for factorization using 16 processors on the NCUBE/2 for a 4 by 4 domain decomposition.

Grid	Sequential time	Parallel time	Speed up	Unknowns
21×21	0.578	0.118	4.90	361
25×25	1.050	0.173	6.07	529
29×29	1.770	0.244	7.25	729
33×33	2.730	0.340	8.03	961
37×37	4.030	0.489	8.24	1225
41×41	5.690	0.659	8.63	1521
45×45	7.730	0.843	9.17	1849
49×49	10.23	1.070	9.56	2209
53×53	13.21	1.397	9.46	2601
57×57	16.78	1.750	9.59	3025
61×61	20.87	2.090	9.98	3481
65×65	25.67	2.460	10.43	3969

running on the Intel iPSC/860, there is a dramatic change in performance. On the 57×57 grid, for example, the speed up drops to 2.78 with the sequential time equal to 1.87 seconds and the parallel time equal to 0.673 seconds. This reduced speed up is caused by the much higher ratio of the communication speed to the computation speed. On the Intel iPSC/860, this ratio is about 10 times higher than on the NCUBE/2 or iPSC/2. For instance, sending one message of length longer than 100 bytes (which, therefore, requires two hand shakings) to only 3 processors in Step 2 takes as long as the time in Step 1 for computing its whole subdomain elimination for a 57×57 grid. This means that the Intel iPSC/860 is a very unbalanced design for such problems as they are of very fine grained parallelism on this machine. We expect that all sparse solvers will be inefficient for these problems on such machines. One runs out of memory long before the problems get large enough to make the slow communication effects negligible.

For comparison, we also mention the performance of our previous nonsymmetric sparse solver PARALLEL SPARSE, on the NCUBE/1 hypercube machine. This code uses throughout the standard fan-out version of Gauss elimination. The parallelism is similar to that of the present algorithm, but it makes use of a dynamic sparse data structure all the way, it also uses information about the non-zero structure of columns (the COL-INFO structure) for organizing communication. The speed up of this algorithm is only about 2.2 for the 37×37 grid problem using 16 processors [18]. We attempted to avoid the cost of manipulating the COL-INFO structure by making a pivot row available to the whole hypercube no matter how many processors actually need it. But this caused a system crash on the NCUBE/1 machine for the test problem due to communication buffer overflow.

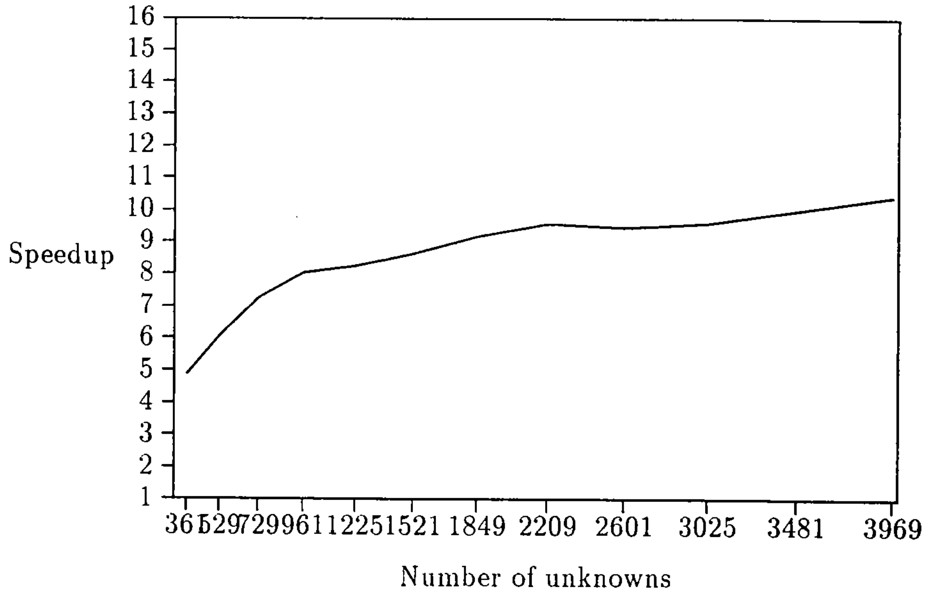

Figure 3: Speed up versus the number of unknowns using 16 processors on the NCUBE/2 for a 4 by 4 domain decomposition.

We have considered only the performance of factorization because it is the major part of solving linear systems. However, we note that achieving good efficiency for parallel back substitution still presents an open challenge. The forward substitution is efficiently handled by incorporating it into the factorization. Previous work [3], [7] and [12] on parallel triangular solvers mainly consider dense matrices. They report that communication costs dominate computation costs even for matrix orders up to 2000. Unless the triangular system is extremely sparse, the sparsity decreases computation costs with little or no decrease in communication cost. Thus the already modest speed ups seen for parallel dense triangular solvers should be expected to be considerably less for parallel sparse triangular solvers. In our example applications, the interface subsystem $L_s \, U_s \, \mathbf{x}_d = \tilde{\mathbf{f}}_d$ to be solved is moderately sparse and of modest size (no more than about 200 unknowns). Direct application of parallelization ideas for dense matrices to this system leads to no speed up at all, the parallel time is about the same as the sequential time. We leave as an open question how to exploit parallelism well for such systems.

5 CONCLUSIONS

We have presented an efficient parallel algorithm for DMMP machines to solve large sparse, nonsymmetric linear systems that arise in PDE applications. The experiments show that high parallelism and good speed up can be achieved for a model problem on machines with a balanced design, such as the NCUBE/2 and the Intel iPSC/2. The key idea is to

use appropriate algorithm components and data structures at different stages for varying sparsity during the elimination process. The idea can also be applied to other applications besides solving PDEs.

We see as the next big challenge the creation of efficient PDE solvers for 3 dimensional problems. The data in Table 1 show that the nature of these matrices is considerably different from those arising from 2 dimensional PDEs. We expect the appropriate sparse solvers to be even more complex as one exploits the special structure of these problems to achieve reasonable efficiency.

Acknowledgements

We would like to thank referees for their many helpful suggestions about this paper and its relation to other methods. These suggestions improved and shortened the presentation of this paper very much. We also appreciate S.C. Eisenstat's useful discussion of the possibility of generalizing the fan-in algorithm to nonsymmetric matrices.

References

[1] Ashcraft, C., S.C. Eisenstat and J.H. Liu, "A Fan-in Algorithm for Distributed Sparse Numerical Factorization", *SIAM J. Sci. Stat. Comput.*, **11**, 1990, pp. 593–599.

[2] Bunch, J.R. and D.J. Rose, "Partitioning, Tearing and Modification of Sparse Linear systems", *J. Math. Anal. and Appl.*, **48**, 1974, pp. 574–593.

[3] Chamberlain, R.M., "An Algorithm for LU Factorization with Partial Pivoting on the Hypercube", Techn. Report CCS86/11, Chr. Michelsen Institute, Bergen, Norway,1986.

[4] Duff, I.S., A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.

[5] Eisenstat, S., Private communication.

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

[7] Eisenstat, S.C., M.T. Heath, C.S. Henkel, and C.H. Romine, "Modified Cyclic Algorithms for Solving Triangular Systems on Distributed-Memory Multiprocessors" *SIAM J. Sci. Statist. Comput.*, **9**, 1988, pp. 589–600.

[8] George, A., "On Block Elimination for Sparse Linear Systems", *SIAM J. Numer. Anal.*, **11**, 1974, pp. 585–593.

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

[10] Houstis, E., J. Rice, and T. Papatheodorou, "Parallel ELLPACK: An Expert System for Parallel Processing of Partial Differential Equations", *Math. Comp. Simul.*, **31**, 1989, pp. 497–508.

[11] Hulbert, L. and E. Zmijewski, "Limiting Communication in Parallel Sparse Cholesky Factorization", *SIAM J. Sci. Stat. Comput.*, **12**, 1991, pp. 1184–1197.

[12] Li, G., T.F. Coleman, "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor", *SIAM J. Sci. Stat. Comput.*, **9**, 1988, pp. 485–502.

[13] Mu, M. and J.R. Rice, "LU Factorization and Elimination for Sparse Matrices on Hypercubes", in *Fourth Conference on Hypercube Concurrent Computers and Applications*, (Monterey, CA, March, 1989), Golden Gate Enterprises, Los Altos, CA, 1990 pp. 681–684.

[14] Mu, M. and J.R. Rice, "A Grid Based Subtree-Subcube Assignment Strategy for Solving PDEs on Hypercubes", *SIAM J. Sci. Stat. Comp.*, 1992, to appear.

[15] Mu, M. and J.R. Rice, "Parallel Sparse: Data Structures and Algorithm", CSD-TR-974, CER-90-17, Computer Science Department, Purdue University, April, 1990.

[16] Mu, M. and J.R. Rice, "The Structure of Parallel Sparse Matrix Algorithms for Solving Partial Differential Equations on Hypercubes", CSD-TR-976, CER-90-19, Computer Science Department, Purdue University, May, 1990.

[17] Mu, M. and J.R. Rice, "A New Organization of Sparse Gauss Elimination for Solving PDEs", CSD-TR-991, CER-90-22, Computer Science Department, Purdue University, July, 1990 (Revised, June, 1991).

[18] Mu, M. and J.R. Rice, "Performance of PDE Sparse Solvers on Hypercubes", *Unstructured Scientific Computation on Multiprocessors*, (P. Mehrotra, J. Saltz and Robert Voigt, eds.), M.I.T. Press, 1991.

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

[20] Mu, M., J.R. Rice and J.W. Wang, "Performance Experiments and Optimizations of PDE Sparse Solvers on Hypercubes", CSD-TR-91-045, CER-91-20, Computer Science Department, Purdue University, June, 1991.

[21] Rice, J.R., "Very Large Least Squares Problems and Supercomputers", in *Supercomputers: Design and Applications*, (K. Hwang, ed.), IEEE Pub. EH0219-6, 1984, pp. 404–419.

[22] Rice, J.R. and R.F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, 1984.

[23] Zhang, X., R.H. Byrd, R.B. Schnabel, "Solving Nonlinear Block Bordered Circuit Equations on a Hypercube Multiprocessor", in *Fourth Conference on Hypercube Concurrent Computers and Applications*, (Monterey, CA, March, 1989), Golden Gate Enterprises, Los Altos, CA, 1989, pp. 701–707.