1990

A New Organization of Sparse Gauss Elimination for Solving PDEs

Mo Mu

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

Report Number:
90-991

A NEW ORGANIZATION OF SPARSE GAUSS ELIMINATION FOR SOLVING PDES

Mo Mu
John R. Rice

CSD-TR-991
July 1990

A NEW ORGANIZATION OF SPARSE GAUSS ELIMINATION FOR SOLVING PDES

Mo Mu*
and
John R. Rice**

Computer Sciences Department
Purdue University
Technical Report CSD-TR-991
CAPO Report CER-90-22
July 1990

ABSTRACT

A new Gauss elimination algorithm is presented for solving sparse, nonsymmetric linear systems arising from partial differential equation (PDE) problems. It is particularly suitable 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 both computational and memory requirements) and sharply reduce the data structure manipulation overhead of standard sparse matrix algorithms. 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.

* 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.

I. 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 organization of sparse Gauss elimination tailored to exploit these structures. 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. 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 prohibitedly 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 [Rice, 1984]) and the considerations studied here for PDE problems are also relevant there.

Symbolic factorization is not well suited for non-symmetric systems as, so far, the techniques generate much too large a data structure. 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 [Mu and Rice, 1990a].

If nested dissection is performed geometrically rather than algebraically (as is natural in PDE problems), 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* [Mu and Rice, 1989a] to some extent.

There are two general row oriented organizations of Gauss elimination focusing on what happens when one eliminates an unknown from a pivot equation. They are *fan-in* and *fan-out* schemes. Another important organization is the *multifrontal* scheme which is unknown oriented (using both rows and columns associated with unknowns). The *fan-in* organization processes the LU factorization row by row as follows.

Algorithm (*fan-in 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.
  - eliminate the k-th unknown (generate and communicate the multiplier vector associated with the k-th unknown)

  end k loop

The alternative is the *fan-out* organization which processes LU by modifying the remaining submatrix using each pivot row.

Algorithm (*fan-out organization*)

- For $k = 1$ to n, do
  - eliminate the k-th unknown (generate and communicate the multiplier vector associated with the k-th unknown)
  - use the k-th pivot row to modify the remaining rows

  end k loop

For a sparse algorithm, the fan-in organization does not process the sparse data structure for a row until it becomes the pivot row and, therefore, only one data structure manipulation is required for each row by using a working buffer. In the fan-out organization the data structure for a row is processed several times, due to fill ins from all previous rows. From this point of view, the fan-in organization is more efficient in manipulating sparse data structures than fan-out.

On a DMMP machine, however, the fan-out organization is more natural because fan-in requires access to previous data (the generated rows of LU) during the whole elimination process. This can create tremendous communication and/or storage requirements. In [Ashcraft, Eisenstat and Liu, 1990] there is a fan-in algorithm for Cholesky factorization of symmetric matrices which forms a modification vector in each processor for a pivot row, say, the k-th row. Unfortunately, for nonsymmetric systems the scalar multipliers for forming the modification vector for the k-th pivot row are those nonzero entries in the k-th column. These are distributed among different processors and thus not available locally, and obtaining them requires substantial communication costs.

It is already observed in [Mu and Rice, 1990b] 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 goal is to use different data structures and algorithmic organizations at different stages in order to increase efficiency. We also use the 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 on 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 II briefly describes a model problem to set the context fo our algorithm. Section III presents the new organization of sparse Gauss elimination, the distributed version is given in Section IV. Section V presents some details on its implementation and Section VI has reports on its performance and comparisons with other algorithms. The final section has comments on performance, extensions and conclusions.

II. THE APPLICATION CONTEXT

The application is described in [Mu and Rice, 1989a] and [Mu and Rice, 1990b]. Thus we are quite brief here. The model linear PDE problem is discretized on a rectangular domain which is divided into $p = 2^{2k}$ subdomains by nested dissection as indicated in Figure 1 for the case $k = 2$. 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 $(2^k n + 2^k - 1)^2$ unknowns in the problem.

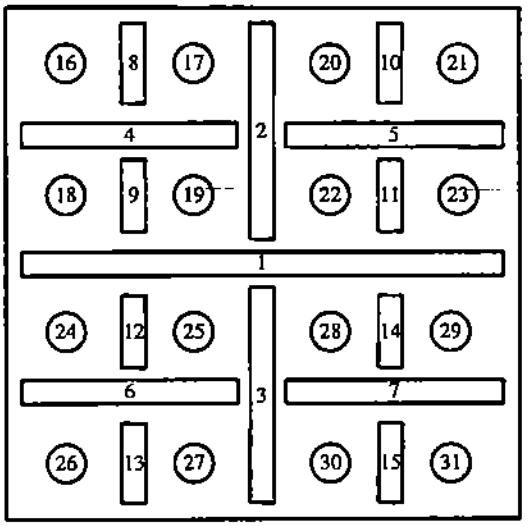


Figure 1. The geometric partition of a rectangle into p subdomains (for $p = 16$) by nested dissection. There are n^2 unknowns in each subdomain and a "line" of unknowns in each separator used to partition the rectangle.

The idea is to order the equations so as to solve the p subsystems on each subdomain, 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 and is of the order $4n + 3$ (in general its order is $2^k n + k - 1$). 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 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 is 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:

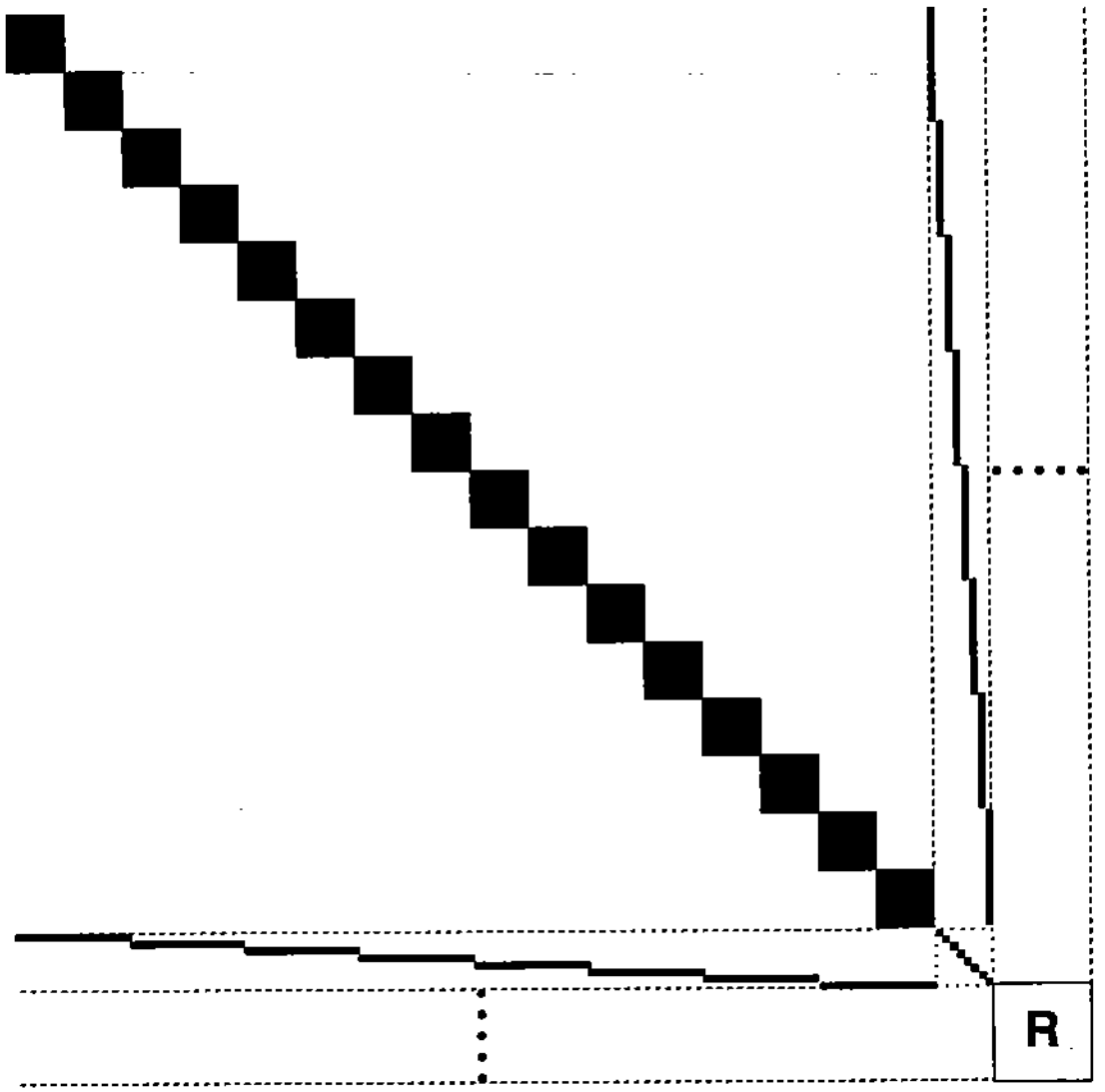


Figure 2. The sparse matrix structure for $p = 16$. For the first two levels the solid boxes are where nonzero matrix elements might be (actually, these blocks are sparse also). The lower right box labeled R (for Rest) is where the diagonal blocks for the other 3 levels are located. There are sparse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

	$p = 16$		$p = 64$		$p = 64$ (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

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 $16 \times (900)^3 / 3 = 4$ 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 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.

III. A NEW ORGANIZATION OF SPARSE GAUSS ELIMINATION

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

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

with

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

The matrices C_i and D contain all the elements of levels 1 to 2^k described in Section II

(see Figure 2). We first perform Gauss elimination on the subdomain equations $A_i \mathbf{x}_i + B_i \mathbf{x}_d = \mathbf{f}_i$ to get

$$\begin{cases} U_i \mathbf{x}_i + \tilde{B}_i \mathbf{x}_d = \tilde{\mathbf{f}}_i, \quad i = 1, \ldots, p \\ \sum_{i=1}^{p} C_i \mathbf{x}_i + D \mathbf{x}_d = \mathbf{f}_d \end{cases} \tag{3.2a}$$

where

$$A_i = L_i U_i, \quad i = 1, \ldots, p \tag{3.2b}$$

are the standard LU factorizations and

$$\tilde{B}_i = L_i^{-1} B_i, \quad \tilde{\mathbf{f}}_i = L_i^{-1} \mathbf{f}_i, \quad i = 1, \ldots, p \tag{3.2c}$$

The last set of equations is unchanged. The corresponding matrix form of (3.1b) is now

$$A = \begin{bmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \ddots & & \\ & & & L_p & \\ & & & & I \end{bmatrix} \cdot \begin{bmatrix} U_1 & & & \tilde{B}_1 \\ & U_2 & & \tilde{B}_2 \\ & & \ddots & \vdots \\ & & & U_p & \tilde{B}_p \\ C_1 & C_2 & \ldots & C_p & D \end{bmatrix} \tag{3.2d}$$

Notice that L_i, U_i and $\tilde{B}_i$ are just the usual parts of the standard triangular factorization of A,

$$A = LU \quad . \tag{3.3}$$

This part of the computation is totally parallel and each A_i factorization can be made

local to each processor. No communication is required at this stage. Therefore, L_i, U_i and $\tilde{B}_i$ can be calculated with full parallel efficiency and cheaply using the most efficient sequential methods. Thus, full parallelism can be expected for this step.

The next step is to eliminate the subdomain unknowns $[\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_p]^T$ from the interface equations. From (3.2a), we have

$$\mathbf{x}_i = U_i^{-1}\tilde{\mathbf{f}}_i - U_i^{-1}\tilde{B}_i\mathbf{x}_d \qquad i = 1, ..., p \tag{3.4a}$$

and

$$\sum_{i=1}^{p} C_i(U_i^{-1}\tilde{\mathbf{f}}_i - U_i^{-1}\tilde{B}_i\mathbf{x}_d) + D\mathbf{x}_d = \mathbf{f}_d \tag{3.4b}$$

So, (3.1a) is transformed to

$$\begin{cases} U_i\mathbf{x}_i + \tilde{B}_i\mathbf{x}_d = \tilde{\mathbf{f}}_i & i = 1, ..., p \\ \tilde{D}\mathbf{x}_d = \tilde{\mathbf{f}}_d \end{cases} \tag{3.5a}$$

where

$$\begin{cases} \tilde{D} = D - \sum_{i=1}^{p} C_i U_i^{-1}\tilde{B}_i \\ \tilde{\mathbf{f}}_d = \mathbf{f}_d - \sum_{i=1}^{p} C_i U_i^{-1}\tilde{\mathbf{f}}_i \end{cases} \tag{3.5b}$$

This may be related to the standard LU factorization by setting

$$\tilde{C}_i = C_i U_i^{-1} \tag{3.5c}$$

Then A may be expressed as

$$A = \begin{bmatrix} L_1 & & & \\ & L_2 & & \\ & & \ddots & \\ & & & L_p \\ \tilde{C}_1 & \tilde{C}_2 & \cdots & \tilde{C}_p \; I \end{bmatrix} \cdot \begin{bmatrix} U_1 & & & \tilde{B}_1 \\ & U_2 & & \tilde{B}_2 \\ & & \ddots & \vdots \\ & & & U_p \; \tilde{B}_p \\ & & & \tilde{D} \end{bmatrix} \tag{3.5d}$$

That is, the $\tilde{C}_i$ are just the appropriate parts of L in (3.3). In standard Gauss elimination one uses the subdomain equations to modify the interface equations in order to get $\tilde{C}_i$ and $\tilde{D}$, and the entries of $\tilde{C}_i$ are the corresponding multipliers.

However, we have observed that there are several disadvantages to complete the factorization for nonsymmetric problems on DMMP machines. First, it is expensive because the interface equations are distributed among different processors and considerable communication is thus involved. The communication requirement here depends on the sparsity of $\tilde{C}_i$ which is much denser than that of C_i because of fill ins. Second, when a pivot equation is received from another processor to be used to modify several interface equations, all these equations have to be processed before the next pivot equation comes. In other words, each interface equation is usually processed many times in an inefficient fan-out pattern according to its relation to the subdomain equations. As our experiments show (see Section V), this requires a lot of time spent in manipulating data structures due to fill-ins. Alternatively, the current pivot equation could be stored in some buffer in an efficient fan-in organization. This is obviously not realistic because it increases the algorithm complexity (perhaps not so important) and storage requirements (very important for current machines). Furthermore, and third, the communication paths here are determined by the nonzero structures of $\tilde{C}_i$, i.e., a processor which holds an equation will determine the destination list of processors for this equation according to the current nonzero structure of the corresponding column in some $\tilde{C}_i$. Since we are interested in nonsymmetric problems, this information cannot be obtained from the nonzero structure of the corresponding row in $\tilde{B}_i$ (unlike in the symmetric case). Because the column structures of $\tilde{C}_i$ are distributed and dynamically updated, an associated data structure (C-INFO) is needed to maintain this information. It also needs to be dynamically updated which requires extra communication which is rather expensive as shown in Section V. These three factors affect the global performance considerably for a DMMP machine. One might consider using a dense matrix data structure for

representing the interface equations. This would save some time in manipulating data structures but it requires storage of about $O(pN^{3/2})$ for the model problem. Even with this, the cost for manipulating the C-INFO data structure is still not avoided.

We now propose a new organization based on the following consideration. Notice that explicitly forming $\tilde{C}_i$ is unnecessary for computing $\tilde{D}$, see (3.5b). If we introduce

$$\begin{cases} \tilde{\tilde{B}}_i = U_i^{-1}\tilde{B}_i \\ \qquad\qquad i = 1, \ldots, p \quad , \\ \tilde{\tilde{\mathbf{f}}}_i = U_i^{-1}\tilde{\mathbf{f}}_i \end{cases} \tag{3.6a}$$

then (3.5b) becomes

$$\begin{aligned} \tilde{D} &= D - \sum_{i=1}^{p} C_i \tilde{\tilde{B}}_i \\ &\qquad\qquad i = 1, \ldots, p \quad . \\ \tilde{\mathbf{f}}_d &= \mathbf{f}_d - \sum_{i=1}^{p} C_i \tilde{\tilde{\mathbf{f}}}_i \end{aligned} \tag{3.6b}$$

To form $\tilde{\tilde{B}}_i$ we only need to perform a row-wise back substitution by solving

$$U_i \tilde{\tilde{B}}_i = \tilde{B}_i, \quad i = 1, \ldots, p \quad . \tag{3.6c}$$

This part of the computation is equivalent to forming the $\tilde{C}_i$ and computing $C_i\tilde{\tilde{B}}_i$ is equivalent to forming $\tilde{D}$ as the modification of the interface equations in the standard LU factorization. Further implementation details are discussed later on in Section IV, but we mention five key points here: First, the data structure for storing $\tilde{C}_i$ and the corresponding manipulations are avoided. Second, the communication required for forming $\tilde{D}$ is now determined only by the nonzero structure of the C_i, which is already represented in the original matrix without any symbolic calculation, and therefore processing the C-INFO data structure is also avoided. Third, the computation of $\tilde{\tilde{B}}_i$ can be performed locally and in parallel without waiting for any information from other

processors. Fourth, from (3.6b) we see that the computations involved in calculating $\widetilde{D}$ can be performed in any order. The inherent sequentiality is moved ahead to the back substitution phase in computing the $\widetilde{\widetilde{B}}_i$. Further, this is a computation local to each processor and thus can be done in parallel. In other words, we reduce synchronization, and thus increase parallelism. And finally, fifth, the communication requirement is reduced because of the following facts. On one hand, the number of destination processors is generally reduced for each pivot row since C_i is sparser than $\widetilde{C}_i$. On the other hand, instead of communicating both U_i and $\widetilde{B}_i$ as in the standard Gauss factorization, we only need to communicate $\widetilde{\widetilde{B}}_i$. If the sparsities of $\widetilde{\widetilde{B}}_i$ and C_i are examined closely (see Section IV), we see that the message volume to be communicated is definitely much smaller.

Because the number of interface unknowns is of a lower order than the total number of unknowns (see Table 1), we propose to represent the D part using a dense matrix data structure. The storage is only $O(N)$ for the model problem where N is the order of the linear system. Therefore, the time in manipulating the symbolic sparse structure for this part is greatly reduced while we can still exploit sparsity to save time in the numerical computation and still exploit the parallelism. As soon as $\widetilde{D}$ is formed, we finally calculate its triangular factorization for the interface submatrix using Gauss elimination, that is

$$\widetilde{D} = L_d U_d \tag{3.7}$$

Of course, further parallelism can be exploited at this stage, for example, by using nested dissection and the corresponding elimination tree for these interface unknowns [Mu, Rice, 1989a]. This is also discussed in Section V.

We now give a complete description of this algorithm as follows.

Algorithm: A New Organization of Sparse Gauss Elimination.

Factorization.

- for $i = 1$ to p, do
  - compute L_i, U_i, $\widetilde{B}_i$ by performing Gauss elimination on subdomain equations.

- compute $\tilde{\tilde{B}}_i$ $(= U_i^{-1}\tilde{B}_i)$ by row-wise back substitution

end i loop.

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

Solution.

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

  end i loop.
- compute $\tilde{\mathbf{f}}_d (= \mathbf{f}_d - \sum_{i=1}^{p} C_i\tilde{\tilde{\mathbf{f}}}_i)$
- compute $\mathbf{x}_d (\tilde{D}\mathbf{x}_d = L_dU_d\mathbf{x}_d = \tilde{\mathbf{f}}_d)$ by forward and 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.

Notice that the solution phase is also highly parallel in the same manner as the factorization.

We may summarize this algorithm as follows. Consider the system

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}$$

where A, B, C, D are generic matrices not related to those used earlier. It has the usual LU factorization as

$$\begin{bmatrix} L_1 & 0 \\ L_3 & L_2 \end{bmatrix} \begin{bmatrix} U_1 & U_3 \\ 0 & U_2 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}$$

We have computed all but L_3 and not changed this part of the matrix at all. It is easily seen that the steps in the algorithm are:

1. Find $L_1, U_1, L_1^{-1}B\ (=U_3), L_1^{-1}\mathbf{f}_1$
2. Factor $\tilde{D} = D - C(U_1^{-1}(L_1^{-1}B)) = L_2U_2$
3. Solve $U_2\mathbf{x}_2 = L_2^{-1}(\mathbf{f}_2 - C(U_1^{-1}(L_1^{-1}\mathbf{f}_1)))$
4. Solve $U_1\mathbf{x}_1 = L_1^{-1}\mathbf{f}_1 - (L_1^{-1}B)\,\mathbf{x}_2$

We see that L_3 is not involved here and the computation can be organized so that it is almost completely done by rows with the only communication between processors being the rows of $U_1^{-1}L_1^{-1}B$, $U_1^{-1}L_1^{-1}\mathbf{f}_1$ and the unknown $\mathbf{x}_2$. Note also that a new problem for a new right side can be solved just as efficiently as if one had saved the standard LU factorization.

We can also relate this organization to *block factorization* [Duff, Erisman and Reid, 1986] by interpreting the following relation

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} A & 0 \\ 0 & I \end{bmatrix} \cdot \begin{bmatrix} I & A^{-1}B \\ 0 & D - CA^{-1}B \end{bmatrix} .$$

The intermediate part $\tilde{\tilde{B}}$ used above is just $A^{-1}B = U_1^{-1}L_1^{-1}B$. As well known, using this relation directly in general is not efficient for sparse matrices because $A^{-1}B$ loses the sparsity of the B part and there is also substantial extra computational cost. In our scheme, $\tilde{B} = L^{-1}B$ is explicitly computed as part of the fan-in Gauss elimination on the subdomain equations without extra cost. It retains most of the sparsity in the B part if a proper indexing is used, i.e., most of top entries in each column of B are zeros. This is accomplished if we index all interior unknowns before boundary layer ones in each subdomain as in [Mu, Rice, 1989b], then each B_i has this property with $\tilde{B} = L^{-1}B = [L_1^{-1}B_1, ..., L_p^{-1}B_p]$. Obviously, the column sparsity of B is preserved because if a column in B is zero then the corresponding column in $\tilde{B}$ or $\tilde{\tilde{B}}$ is also zero. Note that our organization does not explicitly compute $\tilde{\tilde{B}} = A^{-1}B = [\tilde{\tilde{B}}_1, \tilde{\tilde{B}}_2, ..., \tilde{\tilde{B}}_p]^T$, as seen in the next section, it is introduced only as a notation. In computing $\tilde{D}$, only a few of the very last rows in each $\tilde{\tilde{B}}_i = U_i^{-1}\tilde{B}_i$ actualy need to be computed because of the sparsity in the C_i. All this avoids the unnecessary fill-ins and computations in using $A^{-1}B$. Further, in the back substitution phase (steps3 and 4) our organization still

avoids the explicit expression of $A^{-1}B$. We do only one back substitution while computing $A^{-1}B = U^{-1}\tilde{B}$ is essentially equivalent to many such substitutions. In our application, for each subdomain, $\tilde{\tilde{B}}_i$ has m_i nonzero columns, where m_i is the number of interface unknowns on the boundary of the subdomain. Explicitly computing $A_i^{-1}B_i$ is computationally equivalent to m_i back substitutions with the coefficient matrix U_i. For 3-D problems m_i is even larger. [Zhang, Byrd and Schnabel, 1989] consider solving nonlinear systems of block bordered circuit equations using Newton's method. Their Jacobian is sparse with a structure similar to that of Figure 2 and they apply the above block factorization using the explicit $A^{-1}B$ throughout. We believe that the same situation usually exists there with the B part sparse and the number m_i of nonzero columns in each B_i comparatively large, i.e., $m_i \gg 1$. We conclude that our organization is just between the standard Gauss elimination and the block factorization and that it can be applied to more than just PDE sparse matrix problems.

We also observe that if a problem or a machine favors column oriented operations we can similarly devise a version which does not involve U_3 and which keeps B unchanged. The corresponding steps are then

1. Find L_1, U_1, CU_1^{-1} $(= L_3)$, $L_1^{-1}\mathbf{f}_1$
2. Factor $\tilde{D} = D - ((CU_1^{-1})L_1^{-1})B = L_2U_2$
3. Solve $U_2\mathbf{x}_2 = L_2^{-1}(\mathbf{f}_2 - (CU_1^{-1})(L_1^{-1}\mathbf{f}_1))$
4. Solve $U_1\mathbf{x}_1 = L_1^{-1}(\mathbf{f}_1 - B\mathbf{x}_2)$.

In the above, $\tilde{C} = CU_1^{-1}$ is naturally obtained when applying the column Gauss elimination to the first set of columns

$$\begin{bmatrix} A \\ C \end{bmatrix}$$

One computes $\tilde{\tilde{C}} = \tilde{C}L_1^{-1}$, observing that $\tilde{\tilde{C}}L_1 = \tilde{C}$ and taking a transpose to get

$$L_1^T\tilde{\tilde{C}}^T = \tilde{C}^T \quad ,$$

so, we see that this is actually equivalent to a back substitution since L_1^T is an upper

triangular matrix.

IV. THE DISTRIBUTED ALGORITHM

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 $i = 1, 2, ..., p$. 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 [Mu, Rice, 1989b], [George, 1987].

Because all operations on the subdomain equations and the corresponding data structures are row oriented, we prefer to use a row oriented algorithm for solving the matrix equation

$$U_i \tilde{\tilde{B}}_i = \tilde{B}_i, \quad i = 1, 2, ..., p \quad , \tag{4.1}$$

Therefore, the matrix $\tilde{\tilde{B}}_i$ is calculated row by row as follows.

Algorithm (Fan-In Row-Wise Back Substitution)

Let $\tilde{B}_i^{k,*}$, $\tilde{\tilde{B}}_i^{k,*}$ be the k-th rows of $\tilde{B}_i$ and $\tilde{\tilde{B}}_i$, and let $U_i^{k,j}$ be the (k,j) element of U_i which is of order n_i.

- for $k = n_i$ to 1, do

$$\tilde{\tilde{B}}_i^{k,*} = \left(\tilde{B}_i^{k,*} - \sum_{j \in J_i} U_i^{k,j} * \tilde{\tilde{B}}_i^{j,*} \right) / U_i^{k,k}$$

$$J_i = [j \mid k+1 \le j \le n_i,\ U_i^{k,j} \neq 0] \quad , \tag{4.2}$$

end k loop.

This algorithm is of fan-in type as each row $\tilde{\tilde{B}}_i^{k,*}$ is processed only once instead of several times as in a fan-out algorithm. There are two advantages in this. The first is that the data structure for $\tilde{\tilde{B}}_i^{k,*}$ needs to be processed only once. Therefore, this data structure can be either in a dense format or in a compressed sparse format. Second, it is much cheaper to check a whole row $U_i^{k,*}$ for a row oriented data structure than it is to search for each entry $U_i^{k,j}$ each time a particular j is used. In addition, not all rows $\tilde{\tilde{B}}_i^{k,*}$ need to be calculated if careful consideration is taken of the local indexing within each subdomain.

If we separate the interior unknowns from the boundary layer ones for each subdomain and index the former first as in [Mu, Rice, 1990b], then C_i has the following structure

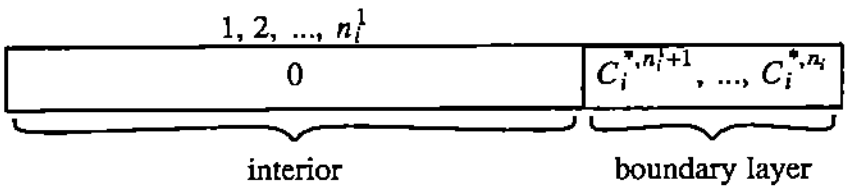


Figure 3. The nonzero structure of C_i by columns.

where the local indices 1, 2, ..., n_i^1 correspond to interior unknowns. The zero block is because the interior unknowns are isolated from the interface unknowns. Therefore, we have

$$C_i * \tilde{\tilde{B}}_i = \sum_{k=n_i^1+1}^{n_i} C_i^{*,k} * \tilde{\tilde{B}}_i^{k,*} \tag{4.3}$$

where $C^{*,k}$ denotes a column of C and $B^{k,*}$ denotes a row of B. So we only have to allocate temporary storage for the rows $\tilde{\tilde{B}}_i^{k,*}$, $k = n_i^1 + 1, ..., n_i$. The total of this storage does not exceed an order of $O(N)$. After the $\tilde{\tilde{B}}_i$'s are used for forming $\tilde{D}$, this storage can be released.

Now we are at the position to describe the distributed algorithm for the new organization of sparse Gauss elimination on a DMMP multiprocessor. The description is for

subdomain i which is assigned to the processor **P**. For each row $\tilde{\tilde{B}}_i^{k,*}$, $k = n_i^1 + 1, \ldots, n_i$ define

$$\text{des\_list}_i^k = \left\{ \begin{array}{c} \mathbf{Q} \mid \text{the interface equations in processor } \mathbf{Q} \text{ have at least} \\ \text{one nonzero entry in the column } C_i^{*,k} \text{ and } Q \neq P \end{array} \right\} \tag{4.4}$$

to be the list of the destination processors **Q** which need to obtain $\tilde{\tilde{B}}_i^{k,*}$ from **P** for computing $\tilde{D}$. Let $n_{\mathbf{P}}^b$ denote the number of rows in $\tilde{\tilde{B}}_j$, $j \neq i$, which need to be obtained by **P** from other processors. These rows can be easily identified from the sparse structure of C_i before the Gauss elimination starts, or even from certain geometric information in discretizing the PDE's. By

$$\text{multicast}(\textbf{message}, \textbf{des\_list}) \tag{4.5}$$

we mean that the **message** is sent to those processors in the destination list **des_list**.

Algorithm: Distributed Parallel New Organization of Sparse Gauss Elimination.

Factorization:

- compute L_i, U_i, $\tilde{B}_i$ by performing Gauss elimination on the subdomain equations.
- for $k = n_i$ to $n_i^1 + 1$, do
  - $\tilde{\tilde{B}}_i^{k,*} = (\tilde{B}_i^{k,*} - \sum_{j \in J_i} U_i^{k,j} * \tilde{\tilde{B}}_i^{j,*}) / U_i^{k,k}$
  - multicast ($\tilde{\tilde{B}}_i^{k,*}$, des_list_i^k)
  - $D := D - C_i^k * \tilde{\tilde{B}}_i^{k,*}$ (for rows of D in **P** and for nonzero operations only)

  end k loop
- for $\ell = 1$ to $n_{\mathbf{P}}^b$, do
  - read a piece of message $\tilde{\tilde{B}}_j^{k,*}$ from buffer (first come, first read)
  - identify indices j and k.
  - $D := D - C_j^{*,k} * \tilde{\tilde{B}}_j^{k,*}$ (for rows of D in **P** and for nonzero operations

only)

end ℓ loop with $\widetilde{D} = D$

- participate in factoring $\widetilde{D} = L_d U_d$ by performing parallel Gauss elimination on the interface submatrix.

Solution:

- compute $\widetilde{\mathbf{f}}_i$ ($L_i \widetilde{\mathbf{f}}_i = \mathbf{f}_i$) by forward substitution
- for $k = n_i$ to $n_i^1 + 1$, do
  - $\widetilde{\widetilde{\mathbf{f}}}_i^k = (\widetilde{\mathbf{f}}_i^k - \sum_{j \in J_i} U_i^{k,j} * \widetilde{\widetilde{\mathbf{f}}}_i^j) / U_i^{k,k}$
  - multicast ($\widetilde{\widetilde{\mathbf{f}}}_i^k$, des_list$_i^k$)
  - $\mathbf{f}_d := \mathbf{f}_d - \widetilde{\widetilde{\mathbf{f}}}_i^k * C_i^{*,k}$ (for elements of $\mathbf{f}_d$ in $\mathbf{P}$ and for nonzero operations only)

  end k loop
- for $\ell = 1$ to n_i^b, do
  - read an entry $\widetilde{\widetilde{\mathbf{f}}}_j^k$ from buffer (first come, first read)
  - identify indices j and k.
  - $\mathbf{f}_d := \mathbf{f}_d - \widetilde{\widetilde{\mathbf{f}}}_j^k * C_j^{*,k}$ (for elements of $\mathbf{f}_d$ in $\mathbf{P}$ and for nonzero operations only)

  end ℓ loop with $\widetilde{\mathbf{f}}_d = \mathbf{f}_d$.
- participate in computing and communicating $\mathbf{x}_d (\widetilde{D}\mathbf{x}_d = \widetilde{\mathbf{f}}_d)$ by distributed parallel forward and back substitutions.
- $\widetilde{\widetilde{\widetilde{\mathbf{f}}}}_i = \widetilde{\mathbf{f}}_i - \widetilde{B}_i \mathbf{x}_d$
- $\mathbf{x}_i = U_i^{-1} \widetilde{\widetilde{\widetilde{\mathbf{f}}}}_i$ by back substitution.

Notice that if a linear system $A\mathbf{x} = \mathbf{f}$ is to be solved with exactly one right hand side $\mathbf{f}$, then the part of computing $\widetilde{\mathbf{f}}_d$ in the solution phase can be merged with computing $\widetilde{D}$ in the factorization phase in a natural way for better efficiency in communication and data structure manipulation.

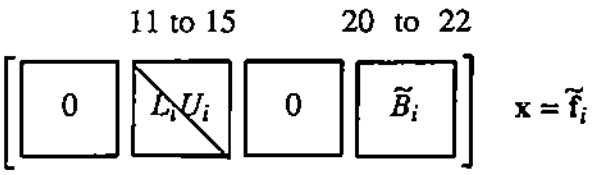
V. IMPLEMENTATION

V.a. Data Structures

For the L_i U_i and $\widetilde{B}_i$ of the subdomain equations, we can use the standard row-wise compressed sparse data structure as follows. The array $\mathbf{a} = a^j$, $j = 1, 2, \ldots, dim(\mathbf{a})$ stores the nonzeroes in the strict triangular parts of L_i, U_i, and $\widetilde{B}_i$ row by row consecutively. The array **ja** identifies the corresponding column indices of entries in **a** where column indices are global to A. The arrays **il, iu, ib** indicate the index positions in **ja** and **a** for the beginnings of rows of L_i , U_i and $\widetilde{B}_i$. Moreover, the index in **ja** and **a** of the first location following the last element in the last row is stored in $\mathbf{il}(n_i + 1)$. That is $\dim(\mathbf{a}) = \dim(\mathbf{ja}) = \mathbf{il}(n_i + 1) - 1$, $\dim(\mathbf{iu}) = \dim(\mathbf{ib}) = n_i$, and $\dim(\mathbf{il}) = n_i + 1$. In addition, an array **diag** with $\dim(\mathbf{diag}) = n_i$ is allocated for storing the diagonal entries in U_i. Thus, the numbers of nonzero entries in the i-th row of the strict triangular parts of L_i, U_i and $\widetilde{B}_i$ are given, respectively, by

$$\begin{cases} \mathbf{iu}(i) - \mathbf{il}(i) \\ \mathbf{ib}(i) - \mathbf{iu}(i) \\ \mathbf{il}(i+1) - \mathbf{ib}(i) \quad . \end{cases} \tag{5.1}$$

For example, assume we have $n_i = 5$ subdomain equations and 3 interface unknowns and the following L_i U_i factorization and $\widetilde{B}_i$:



with

$$\underline{\text{column } 11 \quad 12 \quad 13 \quad 14 \quad 15}$$

$$L_i \backslash U_i = \begin{bmatrix} 3 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 0 & 4 & 2 & 0 & 1 \\ 0 & 1 & 0 & 3 & 2 \\ 1 & 0 & 0 & 3 & 4 \end{bmatrix}$$

$$\underline{\text{column } 20 \quad 21 \quad 22}$$

$$\widetilde{B}_i = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 2 & 0 \\ 1 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

These data are stored in processor **P** as:

		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
a	=	(1,	1,	2,	1,	1,	4,	1,	2,	1,	2,	1,	2,	1,	1,	3,	1,	1),
ja	=	(12,	14,	11,	13,	20,	12,	15,	21,	12,	15,	20,	21,	22,	11,	14,	20,	22),
il	=	(1,	3,	6,	9,	14,	18)											
iu	=	(1,	4,	7,	10,	16)												
ib	=	(3,	5,	8,	11,	16)												

Further compression of the data structure can be made using the technique from the zero-tracking code of the Yale Sparse Matrix Package. The additional arrays **ijl, iju** and **ijb** may be used to indicate sequences of constant values in the array **ja** which can be used for more than one row. We have not used this technique in our implementation because, for our application, the code complexity is increased without too much saving in storage.

One can see from Section III that it is more suitable to use a column oriented data structure for the C part. We use arrays **c, ic** and **jc** to represent C, in a similar manner to what is used above where **c** stores nonzeros in C, **ic** stores row indices (local to the

distributed interface submatrix D in processor **P**), and jc stores the index positions in jc and c of the first locations of the columns of C (this column index is in the *global* sense). For example, assume 2 subdomains (for $i = 1$ and 2) and 5 interface unknowns with the following sparse structure for those rows of C_1 and C_2 in processor **P**:

$$\underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{C_1} \quad \underbrace{\begin{bmatrix} 0 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix}}_{C_2}$$

These are stored in **P** as

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
c =	(1	1	2	1	2	2	1	1	2)
ic =	(1	3	4	2	3	1	2	3	5)
jc =	(1,	1,	1,	4,	6,	6,	8,	10)	

We allocate a standard two dimensional array **d** to store the distributed interface submatrix D in each processor. We let n_d denote the total number of interface unknowns, and $n_{d,i}$ the number of interface unknowns in processor **P**. Finally, we allocate temporary arrays **tb, itb** and **jtb** for the matrix $\tilde{\tilde{B}}_i$ (using the same sparse matrix data structure) and an array **dl** for the destination list of rows of $\tilde{\tilde{B}}_i$ needed by other processors. All the problem data structure arrays, their sizes and the total storage requirements are listed in Table 2 for the model problem from Section II.

Table 2. Storage required for the problem data structures of the i-th subdomain and interface equations assigned to processor **P**. The model problem and notation of Section II is used with p processors, p subdomains, and $N = (\sqrt{p}\,n + \sqrt{p} - 1)^2$ unknowns. The first two columns are the names and sizes of the arrays local to **P** and the final column is the order of the total storage used in all processors.

Array	*Size*	*Order of Total Size*
a	Number of nonzeros in (strict) L_i, U_i and $\widetilde{B}_i$	$N \log_2 N$
ja	Number of nonzeros in (strict) L_i, U_i and $\widetilde{B}_i$	$N \log_2 N$
il	$n_i + 1$	N
iu	n_i	N
ib	n_i	N
diag	n_i	N
c	Number of nonzeros of **P**'s rows of the C	$\sqrt{pN}$
ic	Number of nonzeros of **P**'s rows of the C	$\sqrt{pN}$
jc	$1 + \sum_{i=1}^{p} n_i$	N
d	n_d * $n_{d,i}$	pN
tb	Number of nonzeros in $\widetilde{\widetilde{B}}_i$	pN
jtb	Number of nonzeros in $\widetilde{\widetilde{B}}_i$	pN
itb	$n_i - n_i^1$	$\sqrt{pN}$
dl	$n_{\mathbf{p}}^b$	$p^{3/2}\sqrt{N}$

V.b. Implementation Notes.

When computing L_i, U_i and $\widetilde{B}_i$, any efficient sequential algorithms can be applied because this stage is an absolutely local computation. We implement it in a fan-in manner similar to that in the zero-tracking code of Yale Sparse Matrix Package, i.e., we form them row by row by using previously generated rows since they have been processed and stored in the same processor. Basically, for each working row, our implementation has the initial data of the row from the original matrix and expands this row into the vector dense format using a buffer. Then it makes the necessary modifications on the working row from previous rows. Finally, it compresses the data structure and stores the generated information from the buffer into the storage for L_i, U_i and $\widetilde{B}_i$. The original matrix A can be initially in any form, or perhaps not yet computed explicitly.

Notice that the major computations are done in the buffer, and the major manipulations of data structures are restricted in the last step and performed only once for each working row. The row-wise back substitution for computing $\tilde{\tilde{B}}_i$ is implemented in a similar way with the same advantage. Finally, the operations on the data structure of D is trivial since it is in a dense format. Therefore, no extra cost in manipulating a sparse data structure occurs for this part. However, the parallelism in factorizing D is exploited in the same way as in [Mu, Rice, 1989a, 1990a] and the algorithm organization here is fan-out.

We implement the multicast in the algorithm in Section IV in the trivial way by sending the message to each processor in the destination list as if each pair of processors were physically adjacent. The NCUBE provided primatives **NWRITE** and **NREAD** are used because there are no efficient multicasting routines available so far for the NCUBE. In factorizing D, there are similar multicast tasks but we simply use all processors as the destination lists based on the following considerations. First, the problem at this stage is much denser than before, so the destination lists are much closer the set of all processors. Further, now the information for destination lists cannot be obtained statically in advance. In order to benefit from the efficiency of multicasting, we would have to pay more in manipulating the so called C-INFO data structure. However, these communication tasks are still essentially multicast even though the destination lists are the set of all processors because the broadcast procedure creates a synchronization point among all processors while the algorithm is asynchronous.

VI. EXPERIMENTS AND PERFORMANCE

We report on the performance data for the new sparse Gauss elimination algorithm which is implemented on the NCUBE2 hypercube machine with 64 node processors. For computing speed ups, we use a sequential fan-in Gauss elimination code running on the same node processors, it is believed to be state-of-the-art. Basically, this fan-in code is just the sequential version of the corresponding part in the first stage of the new sparse Gauss elimination used for the subdomain equations, as applied to the whole linear system. The sparse data structure is used all the way. This sequential code is algorithmicly similar to the Zero-Tracking code of the Yale Sparse Matrix Package with the difference in that we do not use the further compressed data structure for indices in the array ju as discussed in Section V.

Our test problem is to solve a PDE using a 37×37 tensor product grid on a rectangle with the five-point star discretization. We use a uniform 4×4 domain

decomposition with interfaces as described in Section II. Table 3 lists some timing results using 16 node processors for the parallel factorization code and one node processor for the sequential one.

Table 3. Timing results (in seconds) for factorization using two sparse matrix codes on the NCUBE2.

Stage	Unknowns	Parallel	Sequential	Speed up
Subdomain	1-1024	0.30	4.7	15.7
Interface	1025-1225	0.59	5.8	9.8
Total	1-1225	0.89	10.5	11.8

In Table 3, the total time for the parallel code is measured by the maximum in all 16 processors used. For a better understanding of the performance of the algorithm, we also list the timing data for the subdomain and interface parts, respectively. The time for processing the subdomain part in the parallel code is measured by the average because the load balance is slightly different among 16 subdomains due to our treatment of the Dirichlet boundary condition. No communication occurs here and we could have had perfect load balancing. The time given for working on the interface part is simply taken as the difference between the total time and the subdomain time. The average time for computing $\tilde{D}$ in the parallel code is about 0.13 second. If the multicast were efficiently implemented, the interface time would also be substantially reduced, the overall speed up, therefore, would be higher. There also is the effect of using a dense data structure for the interface matrix D as mentioned in the introduction. If we implemented the sequential code in the same way, then the interface time might be slightly less than 5.8 due to a simpler data structure, but the difference would not be very big. This is because we still use the fan-in scheme on the interface equations in the sequential case. Then the major computations are performed in a dense vector buffer and data structure manipulations are relatively less important. As noted earlier, this approach is not applicable at this stage for the parallel algorithm. We have run PDE problems with a 23 by 23 grid (441 equations) and 33 by 33 grid (961 equations) using 4 processors and achieved speed ups of almost exactly 4.

For comparison, we also mention the performance of our previous nonsymmetric sparse solver PARALLEL SPARSE on the NCUBE1 hypercube machine. This code uses, throughout, the standard fan-out version of Gauss elimination. The parallelism is

similar to that of the present algorithms, 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 C-INFO structure). The speed up of this algorithm is only about 2.2 for the same test problem [Mu, Rice, 1990c]. We attempted to avoid the cost of manipulating C-INFO 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 NCUBE1 machine for the test problem due to communication buffer overflow.

Finally, we present in Table 4 the speed ups reported in [Ashcraft, Eisenstert and Liu, 1990] using 16 processors for similar PDE problems.

Table 4. Speed ups previously reported for factorization using sparse matrix algorithms applied to PDE problems.

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

All the algorithms in Table 4 are for Cholesky factorization, so no nonsymmetric difficulties as described in the introduction are present. From Table 4 we see that even with an easier problem and larger problem size, the existing algorithms still do not achieve as high a speed up as our algorithm. We have also run problems using 16 processors for a 45×45 grid (1849 unknowns) and achieved a speed up of 13.6.

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 [Chamberlain, 1986], [Li and Coleman, 1988] and [Eisenstat et al., 1988] 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 application the interface subsystem $L_d\ U_d\ \mathbf{x}_d = \tilde{\mathbf{f}}_d$ to be solved is moderately sparse and modest size (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 our open question how to exploit parallelism well for such systems.

VII. CONCLUSIONS

We have presented a very efficient parallel algorithm on DMMP machines for solving large sparse, nonsymmetric linear systems that arise in PDE applications. The experiments show that a very high parallelism can be achieved for a model problem. 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.

REFERENCES

1. Ashcraft, C., S.C. Eisenstat and J.H. Liu (1990), "A Fan-in Algorithm for Distributed Sparse Numerical Factorization", *SIAM J. Sci. Stat. Comput.*, Vol. 11, No. 3, pp. 593-599.
2. Chamberlain, R.M. (1986), "An Algorithm for LU Factorization wit Partial Pivoting on the Hypercube", Techn. Report CCS86/11, Chr. Michelsen Institute, Bergen, Norway.
3. Duff, I.S., A.M. Erisman and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.
4. Eisenstat, S.C., M.T. Heath, C.S. Henkel, and C.H. Romine (1988), "Modified Cyclic Algorithms for Solving Triangular Systems on Distributed-Memory Multiprocessors" *SIAM J. Sci. Statist. Comput.*, Vol. 9, 589-600.
5. 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.
6. Li, G., T.F. Coleman (1988), "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor", *SIAM J. Sci. Stat. Comput.*, Vol. 9, 485-502.

7. Mu, M. and J.R. Rice (1989a), "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.

8. Mu, M. and J.R. Rice (1989b), "A Grid Based Subtree-Subcube Assignment Strategy for Solving PDEs on Hypercubes", CSD-TR-869, CER-89-12, 1989.

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

10. Mu, M. and J.R. Rice (1990b), "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.

11. Mu, M. and J.R. Rice (1990c), "Performance of PDE Sparse Solvers on Hypercubes", *NASA Workshop on Multiprocessors and Non-Uniform Problems in Scientific Computing*, 1990.

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

13. Zhang, X., R.H. Byrd, R.B. Schnabel (1989), "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, pp. 701-707.