1990

The Structure of Parallel Sparse Matrix Algorithms for Solving Partial Differential Equations on Hypercubes

Mo Mu

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

Report Number:
90-976

THE STRUCTURE OF PARALLEL SPARSE MATRIX ALGORITHMS FOR SOLVING ELLIPTIC PARTIAL DIFFERENTIAL EQUATIONS ON HYPERCUBES

Mo Mu
John R. Rice

CSD-TR-976
September 1990

The Structure of Parallel Sparse Matrix Algorithms for Solving Elliptic Partial Differential Equations on Hypercubes

M. Mu[1] and J.R. Rice[2]
Computer Science Department
Purdue University
West Lafayette IN 47907
Technical Report CSD-TR-976
CAPO Report CER-90-19
September, 1990

ABSTRACT

A complete PDE sparse matrix solver consists of several components. Its overall performance strongly depends on their mutual interactions and the effect of application properties, especially when exploiting the parallelism of a distributed memory, message passing multiprocessor. This paper systematically investigates various aspects of the structure and performance of direct methods for solving sparse, nonsymmetric linear systems from PDE applications on hypercube machines. a geometric approach is used to construct and test these PDE solvers. The performance data on the NCUBE are reported which provide the guidance for blending algorithm components to achieve high performance and for creating new, efficient PDE solvers.

[1]Work supported in part by National Science Foundation grant CCR-8619817.
[2]Work supported in part by the Air Force Office of scientific Research grant, 88-0243 and the Strategic Defense Initiative Office contract DAAL03-86-K-0106.

I. INTRODUCTION

The process of solving elliptic partial differential equation (PDE) problems on a distributed multiprocessor can be divided into four major components: **discretization, assignment, indexing** and **solution**. Discretization means that a PDE problem is approximated by a set of discrete algebraic equations. An assignment means that the discrete equations and the associated subtasks are assigned to processors. Indexing means that equations and unknowns are given an order, which establishes a system of linear algebraic equations. Solution means applying a numerical methods (solver) to solve this algebraic system. Potentially, one can apply any kind of algorithm in each of these four components. Various combinations of these algorithms will, of course, have different performances. Our Parallel ELLPACK system provides a test bed for evaluating the performance of such combinations [Houstis, Rice, and Papatheodorou, 1989].

Direct and iterative methods are the two major types of algebraic solution algorithms. We consider the former in this paper, specifically, sparse matrix solvers. There has been a lot of work on developing parallel algorithms for solving sparse systems using Gauss elimination (e.g., [Duff, 1986], [George, Heath, Liu, Ng, 1988], [Mu, Rice, 1989a]). The useful concept of *elimination tree* is first introduced from the algebraic point of view for the sparse structure of matrices. Given a symmetric matrix A, one determines the sparse structure of its Cholesky factor L with $A = LL^T$ by applying symbolic factorization to A and then defines the elimination tree of A from the structure of L with each tree node corresponding to one unknown. This tree reflects the dependency of unknowns during elimination and thus shows the constraints on the order in which unknowns can be eliminated using Gauss elimination. This tree can be used to exploit the parallelisms available in the elimination. If A is nonsymmetric, the definition of the elimination tree is not as natural. One can use traditional elimination trees by doing either a symbolic Cholesky factorization on $A^T A$ (or $A + A^T$) for the coefficient matrix A [George, Liu, Ng, 1988] or a modified symbolic LU factorization with "worse case" assumptions in the fill [George and Ng, 1988]. Note that the shape of this tree directly affects the effectiveness of parallelism in the elimination. A "well shaped" elimination tree should be balanced, wide and short. One can improve tree shapes by reordering the unknowns and equations [Liu, 1988], this is one goal of the indexing component. However, an optimal indexing for the best tree shape to exploit parallelism is in general not optimal for the lowest fill-in which minimizes the arithmetic work. This situation illustrates the general fact that one cannot blindly and independently optimize the four components of a sparse matrix PDE solver. The algorithms must be "blended" together and there is unlikely to be any algorithm for one component that is unconditionly

optimal. The purpose of this paper is to explore the interaction between the algorithms (both old and new) for different components and to provide guidance for the construction of good PDE solvers using sparse matrix methods suitable for particular problems or computing environments. We concentrate on the hypercube architecture, but most of the considerations are applicable to other parallel machines.

The shortcomings of the standard algebraic, sparse matrix approaches for PDE applications are outlined. First, these approaches start with a symbolic factorization for generating an elimination tree. This is appropriate for some applications where many linear systems with the same coefficient matrix and different right hand sides need to be solved. However, it is not appropriate for many PDE software frameworks, such as ELLPACK, where the factorization is normally used only once and where the information generated by the symbolic phase can be much more easily obtained from the physical and geometric properties of the given PDE and the discretization. Therefore, the symbolic factorization is really an extra expense for each run. Second, the above treatments for nonsymmetry are not very efficient for PDE applications. Third, the efforts to develop indexing to balance parallelism and fill-in are too limited in scope, the whole problem solving process must be considered. Finally, and most importantly, it is very hard to appropriately tailor and couple all major components to provide optimal overall performance in solving a given PDE problem. In other words, they may be good approaches for very general and pure algebraic problems, but not as efficient for PDE applications as they do not allow one to easily exploit the special structure of such problems. As argued in [Rice, 1986], linear algebra is not the right model for developing methods for PDEs and it is particularly inappropriate for parallel methods. A similar conclusion is reached in [Chan, 1988] by another line of reasoning.

A geometric (or physical) approach to develop parallel sparse solvers for solving PDE problems is proposed in [Mu and Rice, 1989a], which has several advantages. By extending the conventional elimination tree concept to a block one, we naturally allow non-symmetry in the linear system and totally avoid symbolic factorization. This leads to a well shaped (block) elimination tree, and also allows one to flexibly combine various methods in different regions according to the local properties of the geometry and physics. One is better able to tailor discretization, assignment, indexing and solution components to achieve a satisfactory overall performance. We use this geometric approach in this paper and systematically examine the four major components. Finally, the structure of the implementation appropriate for this approach, as developed for the Parallel ELLPACK system, is presented. It has been implemented on the NCUBE and can be used as a tool for developing and testing algorithms for each of the major

components for solving PDE problems.

I. a. Components of Sparse PDE Solvers on Hypercubes

We start with a listing and brief description of the algorithmic components we may choose to create a parallel sparse matrix solver for elliptic PDEs on a hypercube (or other distributed memory parallel computers). This list probably contains a number of concepts and items which are unfamiliar to most readers. These are described later but we feel it is better to start with a "top down" approach and present a whole picture of the structure first and then fill in the details.

A representative of each of the ten components we identify need not appear in every solver. Further, there is not a unique order in which to select these components, so a "flow chart" or sequential set of steps for the algorithm construction is not appropriate either. Due to the high complexity of the creation of such algorithms, our approach is to choose one general path (our favorite, of course) through these components and discuss various alternatives along the way. Thus we do touch upon all the major choices for components and variations of this path leads to most of the existing algorithms. However, there are algorithms which we do not discuss and we may well be overlooking paths and selections of components that create new algorithms with superior performance characteristics.

For each component we provide a name, a very brief description and a few example algorithmic choices.

1. **Domain Decomposition.** The strategy to divide the domains (or linear algebra problem) into large pieces that are completely or relatively independent of one another. This leads to creating a block elimination tree for the resulting linear system. Example components include:

Nested dissection	*Node amalgamation*
Super nodes	*Square mesh*
Static condensation	

2. **Interface Organization.** The large pieces created by domain decomposition usually have blocks that are completely independent leaving smaller part (*interfaces* or *separators*) of the problem which involves more than one block. This part can be amalgamated into a single system or internally organized in various ways. Examples:

Nested dissection separators	*Capacitance matrix*

3. **Matrix Orientation.** The matrix problem can be organized in several ways. The PDE source of the matrix problem makes organization by rows the most desirable and, to simplify things a little, we only consider rows here. Within these organizations there are various choices for specific sparse matrix data structures. Examples:

Rows *Columns* *Blocks*

4. **Discretization.** This is the technique used to approximate the continuous PDE problem by a linear system. Examples:

Ordinary finite differences *Collocation with cubic splines*
High order differences *Galerkin with Hermite cubics*

For simplicity, we assume all these lead to linear systems with the same "general" sparse matrix nature.

5. **Parallelism Organization.** Parallelism can be exploited at four different levels:

5-1: **Subdomains.** The completely independent pieces created by domain decomposition may be processed in parallel.

5-2: **Within Interface.** Groups of equations at the nodes of the elimination tree (with various sets of interface equations) often include some independent equations even after the 5-1 eliminations are complete.

5-3: **For One Unknown.** The elimination of an unknown can be carried out independently in all the equations containing it.

5-4: **Within an Equation.** The operations on an equation used to eliminate an unknown in it usually involve many independent tasks.

The work done in parallel at these levels can be organized in many ways, often intermixing the various levels. Examples:

Fan-out *Fan-in* *Multi-frontal*

6. **Nodal Indexing.** Within each node the indexing (elimination order) can be chosen in many ways. At the subdomain level (parallelism 5-1) all the sequential indexings can be considered. In the separator or interface groups (nonleaf nodes of the elimination tree) many of these indexing can still be applied. Examples:

Minimum degree *Nested dissection* *Random*

7. **Node Assignments.** The equations at the nodes of the elimination tree are assigned to processors (and their memories) in a fashion intended to minimize communication costs. Examples:

Subtree-subcube *Grid-based subtree-subcube*

8. **Internal Node Assignments.** Nodes at higher levels of the elimination usually have several processors assigned to them. The equations within a node must then be distributed among these processors. Examples:

Local wrapping *Grid-based*

9. **Communication Protocol.** Information between processors can be sent at various times; the objective is not to delay other processors nor to spend too much effort in communication. Examples:

Write as soon as possible *Pipelined*
Read as soon as possible *Read as late as possible*

10. **Back Solve.** Almost all the work is in factorizing the coefficient matrix and almost every thing in the previous nine components is oriented toward this task. The back solve phase of solving the PDE must start with whatever is left from the factorization. Since it is very difficult to exploit parallelism in the solution of sparse triangular linear systems, this phase is relatively much more expensive for parallel computation. Examples:

Fan-in *Fan-out* *Copied blocks* *Cyclic*

Finally, we note that the path we choose in selecting components is based in certain directions due to the PDE source assumed for the sparse matrix problem.

* Pivoting is not required because the problem is elliptic. For some discretizations it may be necessary to use small diagonal blocks as "pivots" rather than single matrix elements.
* The problems are non-symmetric. The lack of symmetry can arise from the PDE itself, from the discretization used, from the domain, or from the boundary conditions.

II. A GEOMETRIC APPROACH TO DEVELOP PARALLEL PDE SOLVERS

II. a. Domain Decomposition and Block Elimination Trees.

A geometric (or physical) approach based on domain decomposition to develop parallel PDE solvers is briefly described in this section. For simplicity, we consider a PDE problem on a rectangular domain Ω. The approach can be extended easily to general domains. Suppose we have $p(= 2^{2d})$ processors available, 2^d in each direction. By a square mesh partition Ω is divided into p subdomains Ω_{ij}, $i, j = 1, 2, ..., p^{1/2}$ as shown in Figure 2.1. One puts a local grid on each subdomain Ω_{ij} and discretizes the local problem whose solution U_{ij} only depends on unknowns at grid points of $\partial\Omega_{ij}$, the boundary of Ω_{ij}. These unknows are the *interface unknowns*.

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

Figure 2.1. Square mesh domain partition of a rectangle into $p = 2^{2d}$ subdomains for $d = 2$.

Initially, all interior unknowns U_{ij} are eliminated locally as in the standard domain decomposition approach. This step is obviously totally parallel among all subdomains. Then, all processors participate in eliminating interface unknowns. To exploit parallelism better, we use dissection in alternating directions to partition the interface set into several levels suitable for a hypercube machine, each level consists of several **separators**, groups of unknowns which separate regions. The partition, which we call the **one way nested dissection decomposition**, is shown by Figure 2.2 with circles representing the unknowns interior to the subdomain Ω_{ij}, the boxes representing the **separators**. For simplicity, they are all called **subdomains** of this domain decomposition and are numbered from top level to bottom level as shown in Figure 2.2.

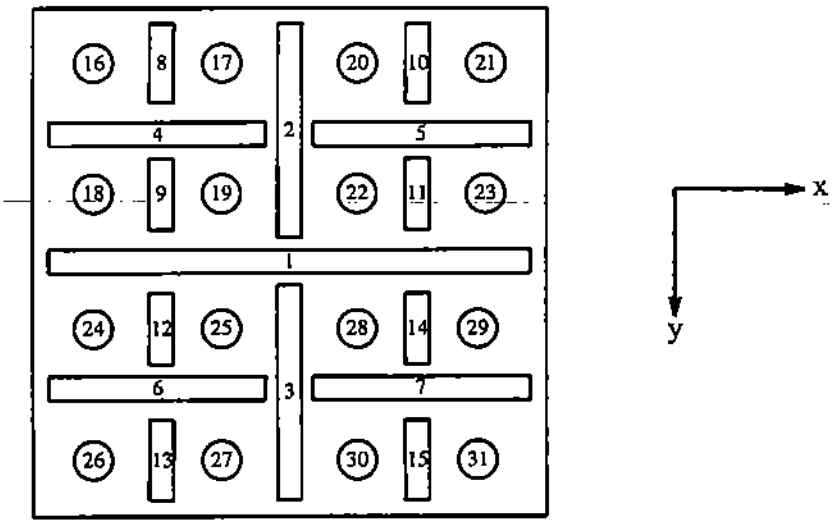


Figure 2.2. Partition of the subdomain interfaces in Figure 2.1 using one way nested dissection. The circles (16–31) represent the 16 groups of interior unknowns and the boxes represent groups of interface unknowns or separators. All boxes of the same size are on the same level of the block elimination tree. (See Figure 2.3.)

This domain decomposition naturally inherits certain parallelism because the PDE discretization process leads to a local or boundary dependence property for interfaces. For example, if we consider the union of subdomains 16,17,8 as a more general subdomain Ω'_8 then the local interior solution set U'_8 is uniquely determined by unknowns on $\partial\Omega'_8$. This relation holds similarly for groups at higher levels of the dissection decompositions. This local dependency can be described by a binary tree as shown in Figure 2.3 with each tree node corresponding to a subdomain in the decomposition as shown in Figure 2.2. We call it a **block elimination tree** because it plays a role similar to that of the standard **elimination tree** in exploiting parallelism. However, each node now corresponds to a block of unknowns and equations, rather than a single unknown and equation. In addition, the tree has an excellent shape suitable for exploiting parallelism. Thus, the top level data structure preserves the symmetry and balance derived from the nested dissection domain decomposition even though the linear system itself might not be symmetric and its indexing might not be exactly nested dissection. Such lack of symmetry occurs in PDEs from, for example, (a) lower order derivative terms, (b) mixed boundary conditions, (c) irregular geometry, or (d) nonsymmetric discretization such as collocation with bicubic Hermite piecewise polynomials.

Discussion of the indexing issue is deferred to in Section IV. In summary, we are seeking the parallelism in the block sense no matter what local properties the linear system has locally. This block elimination tree has the following properties: (a) Each node corresponds to a set of unknowns from one location, local ordering and lack of symmetry in the linear system do not affect the tree structure. (b) Eliminating a node only has effects on its ancestors. (c) The elimination of nodes that are not descendants/ancestors of one another are independent of one another. It thus implies several basic rules for assignment, indexing and solution to exploit the full parallelism inherent in the block elimination tree.

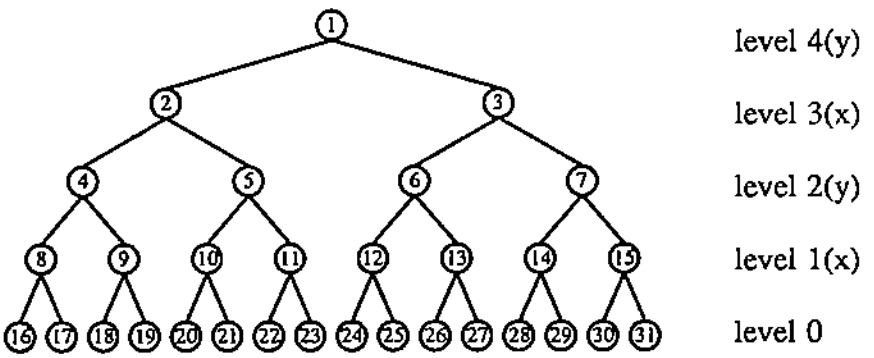


Figure 2.3. Block elimination tree produced by one way nested dissection domain decomposition. The numbering of nodes corresponds to the groups of unknowns in Figure 2.2, the x, y levels refer to the directions of the bisection.

This approach gives a block elimination tree with a very simple data structure, one does not have to store it and its manipulation is also very efficient. For cases arise from standard nested dissection or a nonrectangular domain where the trees are not completely binary, they can be handled by introducing some empty nodes as described in [Mu, Rice, 1990a].

This block tree formulation includes or is closely related to previous similar approaches, such as the **node amalgamation** in multifrontal methods [Duff, 1986], the **supernode** in symbolic factorization and the compute-ahead, fan-in Cholesky factorization [Liu, Ng and Peyton, 1990] as well as in sparse factorization on workstations [Rothberg and Gupta, 1990], and even earlier, the **static condensation** technique in finite element methods. Note that this approach allows general local irregularities

during the whole PDE solution process, such as the lack of symmetry in the elimination tree representation.

II.b. Parallelism

There are four kinds of potential parallelism in these problems. First, elimination steps in independent nodes (nodes not ancestors/descendents of one another) of the block elimination tree can execute simultaneously. To see this, consider two independent nodes S and T. Property (b) above says that eliminating S and T will not affect with each other, while property (c) means that the effects on their common ancestors, if any, are also independent. Therefore, we can independently start to eliminate a node as soon as all of its descendants have been eliminated. We call this the **outer parallelism**. Second, if there are several processors available for a single node, we can also exploit **inner parallelism** within the node. This does not occur at leaf nodes if each leaf node has only one processor as is usual. For the other nodes we can apply various efficient parallel dense solvers to exploit the inner parallelism. Third, the tasks to modify an equation to eliminate an unknown (or simply, a *modification*) are independent for different equations, just as for dense matrices. Finally and fourth, modifications, even on the same equation, due to independent descendant nodes can be performed in arbitrary order and hence in parallel. Different ways of exploiting these kinds of parallelisms in an algorithm organization generate different parallel algorithms. Example organizations include **fan-out** [George, et al., 1988], **fan-in** [Ashcraft, et al., 1990], **multifrontal** [Duff, 1986], and a **new Gauss elimination organization** [Mu, Rice, 1990b]. Some of these are discussed in more detail in Section V.

III. DISCRETIZATION

By discretization, we mean both discretizing a geometric domain by a grid or elements and approximating a PDE operator by a set of discrete algebraic equations. Potentially, local grids (geometric discretization) and local discrete systems (PDE discretization) can be different depending on the local geometric and physical properties. In other words, each tree node can have its own discretization. For example, with a nonrectangular domain Ω, a triangular subdomain would have a triangular finite element grid while a rectangle subdomain could have a simpler tensor product grid. Even with a rectangular domain Ω and a uniform domain decomposition one might put coarser grids on subdomains near $\partial\Omega$ for the sake of load balancing [Zmijiwski, 1989] or put finer grids on certain subdomains where some physical singularities are presented. As to PDE discretization, one might apply a *Galerkin finite element* scheme

[Strang, Fix, 1973] on certain subdomains where the local physical problems are well presented in an energy form, a *finite difference* or *collocation* scheme [Forsythe, Wasow, 1960], [Houstis, Mitchell, Rice, 1985] on subdomains where the PDE operator is in a more general form. For properly coupling these local algebraic systems on the interfaces of subdomains, see [Rice, 1989].

IV. ASSIGNMENT

By assigning an unknown to a processor we mean assigning both the problem data and the computation subtask associated with this unknown. From the algebraic point of view, these are basically row-wise assignments. To achieve high parallelism, load balancing and low communication costs, it is generally required to (a) avoid assigning independent nodes to the same processor, and (b) assign processors to a single node so as to have minimal communication connections. The standard wrapping assignment is not as effective here even though it achieves load balancing.

In [George, Liu, Ng, 1987] an attractive **subtree-subcube with local wrapping** assignment is proposed. We refer to it as the *standard* subtree-subcube assignment. It is a top to bottom process. First, the root node of the elimination tree is assigned to the whole hypercube and then the hypercube is split into two subcubes to which the two descendent subtrees are assigned. This process goes on recursively until all subtrees become assigned to single processors. The assignment within each node is simply in wrapping manner. Note that: (a) eliminating an unknown in a node need not affect all of its ancestor nodes, and (b) even when effects occur in some ancestor nodes, they need not affect all equations in them. Geometrically, the effect of elimination spreads in a multifrontal manner with each processor starting with one subdomain and continuing to work on "merged" domains containing it as interface (separator) unknowns are eliminated. However, one cannot represent these properties completely by elimination trees and yet they may affect the parallel efficiency very much. This suggests that unknown assignments be made in a multifrontal manner with a processor responsible only for those unknowns located at the fronts of some nodes to which the processor has been assigned. If several processors (usually a subcube) correspond to the same set of unknowns, then local wrapping can be applied within this set. This is made more precise below.

We define the **grid based subtree-subcube** assignment as follows. It is a bottom to top assignment process. Let us denote the levels in the block elimination tree from bottom to top by 0, 1(x), 2(y), 3(x), 4(y), $\cdots$ as in Figure 2.3. The x and y refer to the horizontal (j or x) and vertical (i or y) directions, respectively. The process is

related to Figure 2.2 where level zero consists of the leaves, $1(x)$ consists of the eight separators 8-15, $2(y)$ consists of the 4 separators 4-7, $3(x)$ consists of separators 2 and 3, and $4(y)$ consists of separator 1. The first step is to map the given hypercube to a two dimensional grid (for domain decomposition) by the well-known *gray code* such that adjacent processors are directly connected and Ω_{ij} is assigned to processor P_{ij}. This defines the assignment at the leaves of the block elimination tree, the 0 level. Next, we subdivide each separator on the $(i+1)^{st}(x)$ level and the i^{th} (y) level into $2^{i/2}$ segments. This subdivision of separators corresponds to the natural geometric segments along the separators of the domain decomposition. We take care of the intersection points of adjacent segments in each separator by adding an intersection point to its top (left) segment for the $x(y)$ direction. Then we assign each segment on the i^{th} level to the closest processors as follows: for i odd (x direction) use $2^{(i+1)/2}$ processors, for i even (y direction) use $2^{i/2}$ processors. The assignment within each segment uses wrapping. This scheme is illustrated in Figure 4.1. Thus, processors P_{11} and P_{12} are assigned to the interface unknowns of separator 8 (the upper left box in Figure 4.1). For comparison Figure 4.2 illustrates the standard subtree-subcube assignment strategy.

The potential of this assignment for reducing communication is seen by examining separator 4 which is the top, left horizontal box in Figures 2.2, 4.1 and 4.2. In Figure 4.1 we see the unknowns of seperator 4 divided into two separate groups (segments). In the grid based subtree-subcube assignment (Figure 4.1), the processors P_{11} and P_{21} only handle the interface between the two subdomains (16 and 18) they handled at the lower level. In the standard subtree-subcube assignment (Figure 4.2), the processors P_{11} and P_{21} are part of a group of processors handling the four subdomains (16, 17, 18 and 19). Thus P_{11} and P_{21} must now obtain information about subdomains 17 and 19 at this step while this is not required in the grid based subtree-subcube assignment. This reduction in the communication occurs in a similar manner for every separator.

In [George, Liu, Ng, 1987] it is proved that the total amount (or volume) of communication for the standard subtree-subcube assignment is $O(pN)$. This order is optimal in the sense of minimizing traffic volume for nested dissection algorithms, our grid based subtree-subcube assignment has the same optimal order as the analysis in [George, Liu, Ng, 1987] applies for all types of subtree-subcube assignments. We give an analysis in [Mu, Rice, 1989b] which provides estimates for the communication complexity of start ups as well as of traffic volume for both assignments when applied to a modified nested dissection indexing and fan-out organizations as described in Sections 5 and 6, respectively. The grid based subtree-subcube assignment gains an additional $O(log_2 p)$ reduction in start up cost compared to the standard assignment, and that it

achieves the optimal order of start ups for the nested dissection algorithms. As a by-product, this assignment also reduces the volume of communication by a factor of about two. For more details, see [Mu, Rice, 1989b].

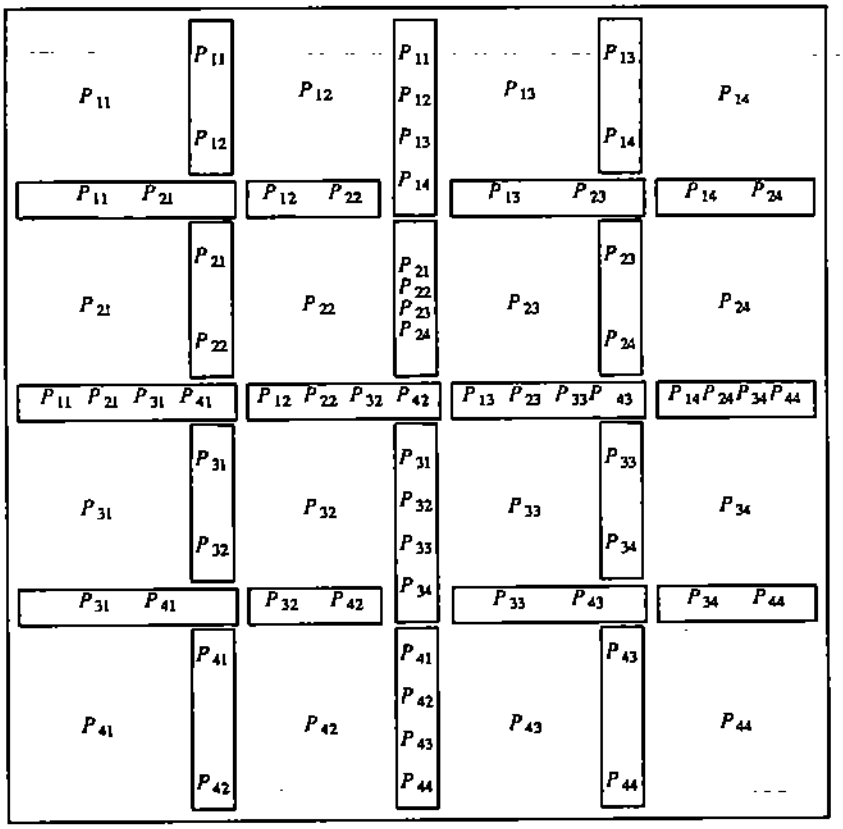


Figure 4.1. Grid based subtree-subcube assignment for 16 processors. Within the subdomain interfaces we show how the processors are assigned to unknowns in parts of the separators.

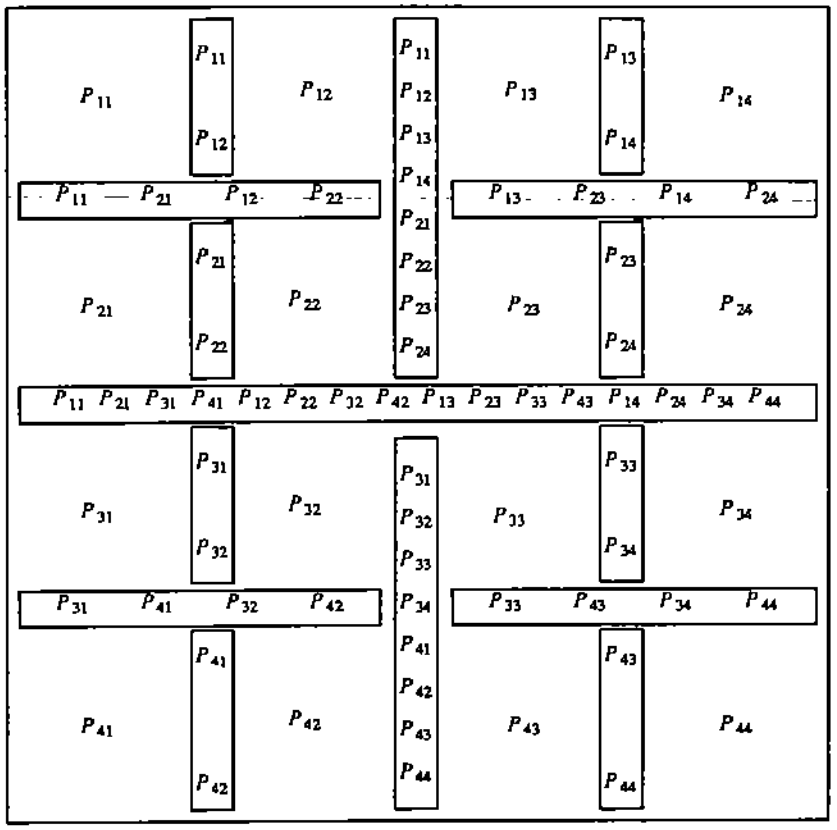


Figure 4.2. Standard subtree-subcube assignment for 16 processors. Within each box unknowns are assigned in wrapping manner to processors shown in the box.

V. INDEXING

Two important characteristics of a parallel sparse solver are the parallelism and the fill-in, both of which depend on the indexing used. On a message-passing, distributed memory machine, the amount of communication involved is also a very important characteristic. The **nested dissection** ordering generates well shaped elimination trees and preserves moderately low fill-in and is thus currently widely recommended for parallel sparse computations. However, we notice that there exist orderings, such as **minimum degree**, which may have less fill-in than nested dissection does [Liu, 1989], especially for irregular domains where nested dissection is not as efficient. But these fill-in oriented orderings often generate poorly shaped elimination trees and are therefore less well suited for parallel computation. Several questions are appropriate here. First, are there indexings which preserve good parallelism and which have fill-in lower than nested dissection? Second, we expect nested dissection to be efficient primarily

for regular, or nearly rectangular domains of solving the PDE. How should one modify the nested dissection domain partitioning strategy for more general domains? Third, is the communication cost for nested dissection nearly minimal?

We know of no work on the first question but conjecture that the fill-in using nested dissection is rather low. There is extensive work related to the second question. Many people are using geometry based heuristic methods for PDE problems, see [Chrisochoides, Houstis, Papachiou, Kortesis and Rice, 1990] for an example and references to other recent literature. There is some related work from an algebraic approach, for example, [Liu, 1988] improves the parallelism using rotation. We believe the algebraic approaches are not well suited for this problem in PDE applications compared to the geometric approaches. Most efforts related to the third question involve assigning the processors well. The *multifrontal* idea described in the last section is an attractive geometry based approach as one can visualize keeping the elimination fronts separated and thus not requiring as much communication. We see that the goals of low fill-in, high parallelism and low communication require blending the algorithms in order to preserve their best features. In the remainder of this section we illustrate how to create an indexing by blending ideas from the nested dissection, minimum degree and multifrontal methods. We then show how to further tailor the indexing to improve load balancing. From now on we generically use **minimum degree** to mean whatever indexing generates the lowest fill-in to a given set of unknowns. The classic minimum degree is not necessarily the only choice because it is a heuristic algorithm and sometimes is not the best one.

V.a. The Combination of Nested Dissection, Minimum Degree and Multifrontal Techniques

The one way nested dissection domain decomposition insures that the parallelism is as high as that of standard nested dissection and it keeps the fill-in from eliminating interface unknowns fairly low. It retains the advantage of nested dissection at the stage of eliminating interface unknowns because it keeps the number of them to be $O(N^{1/2})$, a lower order compared to the number N of total unknowns. We believe that parallelism is more important than fill-in at this stage. However, we do not perform the dissection all the way on the linear system (using **incomplete nested dissection**) in order to minimize its disadvantages. Recall that we are interested in the case where p, the number of processors or subdomains Ω_{ij} is much less than the number N of unknowns. Therefore, performing the nested dissection domain decomposition is both more efficient and easier than doing the complete (or standard) dissection of the linear

system. Further, because each Ω_{ij} is usually assigned to a single processor, the elimination within Ω_{ij} is, therefore, essentially a sequential computation. So fill-in is more important than parallelism at this stage and we should use a minimum degree indexing for reducing fill-in. However, there is also communication involved at this stage even though it is computationally sequential within each subdomain. Both nested dissection and minimum degree are not good in this respect since they make the elimination fronts involve interfaces at the very beginning. To minimize communication at this stage, we apply a **multifrontal** scheme as shown in Figure 5.1.

Denote the boundary layer of grid points in Ω_{ij} (those next to $\partial\Omega_{ij}$) by B_{ij}. Without loss of generality assume that only unknowns on B_{ij} are related to those on $\partial\Omega_{ij}$ in the linear system, such as **five point star** would generate. We first eliminate the unknowns on the interior of Ω_{ij} using a minimum degree ordering. There is no communication required at this stage. Then the unknowns on B_{ij} are eliminated. These unknowns and equations are linked to the interface unknowns in the separators so inter-processor communication is involved. For some other PDE discretizations in Ω_{ij}, the boundary layer B_{ij} will be thicker but it is still relatively thin.

This new indexing exploits the advantages of all techniques involved. For fill-in, it achieves the same order as the optimal indexing scheme would, (such as *minimum degree* used locally in Ω_{ij}) because the order of the fill-in generated by the $O(N^{1/2})$ interface unknowns is not worse than $O(N)$. Its parallelism is comparable to that of nested dissection since it is essentially sequential in each Ω_{ij}. It also reduces the communication start ups to an order of $O(N/p)$ compared to the standard nested dissection and, together with our new grid-based subtree-subcube assignment, achieves the optimal orders of communication for both start ups and message volume [Mu, Rice, 1989b].

If the hardware favors collecting messages for bulk communication, then this approach is compatible as all the boundary lines are separately identified and easily associated with communication destinations. For improving communication efficiency, Zmijiwski proves that the fan-in organization ([Zmijiwski, 1989], [Ashcraft, et al., 1990]) achieves the same effect as our multifrontal variation of indexing. However, the fan-in scheme can only be applied to symmetric matrices as pointed out in [Mu, Rice, 1990b]. On the other hand, [Liu, 1989] uses an idea similar to that presented above to reduce the sensitivity of **minimum degree** to the initial matrix ordering by mixing the classic **minimum degree** with **nested dissection**. Liu remarks on its application in parallel computation; he does not consider the multifrontal approach.

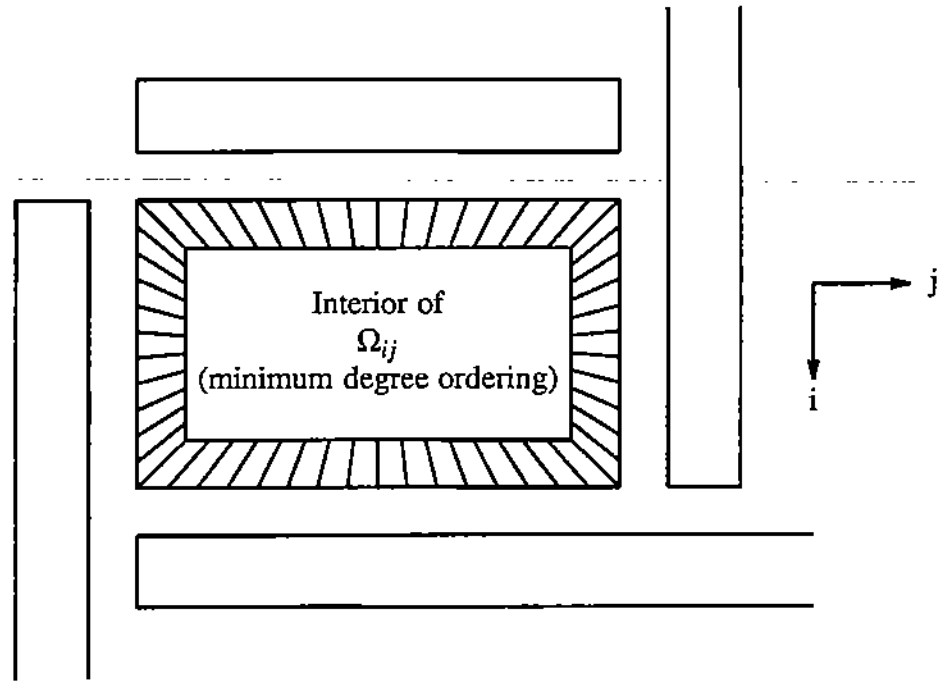


Figure 5.1. View of a typical subdomain Ω_{ij} of the domain decomposition. In the PDE discretization there are k^2 unknowns (grid points) in Ω_{ij}, $(k-2)^2$ in the interior and $n_0 = 4(k-1)$ in B_{ij}, the boundary layer (dashed area). The separators (shown as boxes) are lines of grid points separating the subdomains.

V.b. Local Indexing for Seperators and Local Balance

The local indexing in the interface separators has little effect on computation and fill-in since the local problems are almost dense by the time elimination comes. With a subtree-subcube assignment it also does not affect communication much. The simplest local indexing in a separator is a geometric wrapping. But, as we have seen in Section IV, the assignment within each separator does affect communication substantially and the grid-based subtree-subcube segment-wise assignment strategy achieves the optimal communication order. If we couple this assignment with the local geometric wrapping indexing, then the assignment for the local dense problem in each separator is handled algebraicly in a block manner. It is well known that the algebraic wrapping assignment achieve better load balance for a dense system than a block one does [Geist, Heath, 1986]. Therefore, we also need to modify the local indexing in order to couple the efficient assignment strategy and to achieve a good local balance. One simple remedy is to do local indexing also in a segment-wise manner similar to that of the assignment

as illustrated in Figure 5.2 and its performance is shown in Section VIII.

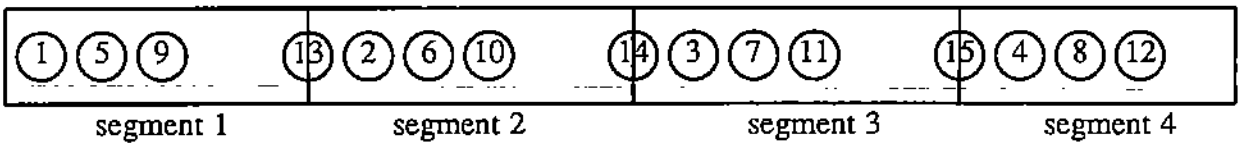


Figure 5.2 The segmentwise-wrapping local indexing in a typical separator.

VI. SOLUTION

VI. a. Factorization

The potential parallelism inherent in the block elimination tree provides many ways to develop parallel algorithms. While the fan-in organization is efficient for data structure manipulation and communication, it is essentially restricted to symmetric matrices or a shared-memory environment. Therefore, we mainly consider fan-out schemes. We first discuss the commonly used **pipelining** idea whose philosophy in the context of sparse matrix algorithm is "as soon as something needed by another processor is computed, send it off so the other processor will not have to wait on it". This is clearly a good idea for increasing the effectiveness of parallelism as it reduces synchronization delays. If we paraphrase this idea as "write as soon as possible" we see that there is also a corresponding strategy for reading, namely, anything between the extremes of "read as soon as possible" to "read as late as possible". Abstractly, the "read as late as possible" strategy should be preferred for this, again, minimizes the synchronization delays. That is, a processor should keep busy computing the results that it and others need so others will have to wait as little as possible on it.

A high level algorithmic description of a pipelined distributed sparse algorithm of LU factorization by Gauss elimination for a processor P, as in the module PARALLEL SPARSE, is as follows.

```
for level l from bottom to top, do:
    for each node S, with equations assigned to P, on level l, do:
        if l = 0, then
            elim_local(S)
        else
```

```
            elim_global(S)
            elim_local(S)
          endif
        endif
      end of S loop
    end of l loop
```

The procedure **elim_local(S)** is where processor P participates in eliminating unknowns in S by performing the associated modifications on equations assigned to P. For those equations of S assigned to P, it also calculates the corresponding multiplier vectors and sends them to other processors. When level $l = 0$, one usually assigns S to only one processor and in this case elim_local is a sequential sparse solver. Otherwise, it is a sort of parallel dense solver using processors assigned to S.

The procedure **elim_global(S)** is where processor P performs the modifications on its equations due to eliminating unknowns in the descendants of S in the block elimination tree which have no equations assigned to P. Therefore, the effects of elimination at these nodes have not been processed by P unitl **elim_global(S)**. The pipelining idea applied here is for P to start **elim_local(S)**, and send the multiplier vectors to other processors as early as possible. It only processes the modifications in **elim_global(S)** due to eliminating unknowns in the descendants of S without waiting for the completion of eliminating other nodes independent of S, even though the associated modifications are ready for this. This is pipelining in the block sense. Within **elim_local(S)** a similar pipelining idea, as in dense solvers (e.g., [Geist, Heath, 1986]), can also be applied. In this case, if P has several equations to be modified by a pivot row it first processes the first one among them, then checks if that row is the next pivot equation, and if so, send it out immediately. It then resumes completing the modifications on the remaining equations.

For a subtree-subcube type assignment, there is an *elimination path* for each processor P in the block elimination tree from bottom to top with exactly one assigned node S_l on each level l for $l = 0, 1, ..., L$. Therefore, the general sparse algorithm can be simplified as follows.

```
elim_local(S_0)
for l = 1 to L, do:
    elim_global(S_l)
    elim_local(S_l)
```

end of *l* loop

Ideally, a processor should interrupt its computation to read messages only when it actually needs the message. However, one observes that those authors who say what they are doing in this regard tend to use the "read as soon as possible" strategy. We believe the reason for this apparently illogical choice is as follows. The combination of "write as soon as possible" and "read as late as possible" means that messages accumulate somewhere (in system buffers) between the writing and reading. These buffers are not huge (one rarely can discover their size easily, they are a little bit larger on the NCUBE 2 than the NCUBE 1), say 25,000 bytes, and when they overflow the results are usually unpredictable, drastic, and hard to diagnose. Thus the "read as soon as possible" strategy becomes attractive even though it does increase synchronization delays. Just how much the delay is increased is hard to measure, but it is surely some.

The primary trouble with buffers currently may be their implementations, but the size problem is inherent in the computation. A problem with 256 subdomains, each with a 100 by 100 grid has about 100,000 subdomain boundary points which result in messages being sent. If the computation were perfectly balanced, all these messages would be sent before any was read. Recall that a single message may go to many processors and it is many bytes along. The system communication buffers would require several hundred megabytes of memory, so the "read as late as possible" strategy for pipelining is unlikely to be useful for sparse matrix algorithms on hypercubes. However, it is plausible that something better than "read as soon as possible" can be devised. Of course, a system option (hardware implemented) for a "full buffer interrupt" could lead to both efficient computation and modest buffer size. We also notice that recently [Ashcraft, Eisenstat, Liu, Peyton and Sermen, 1990] proposes a **compute-ahead fan-in Cholesky scheme** to reduce the synchronization delays in the "read as soon as possible" strategy.

To maximize the potential pipelining for a given problem on a machine with a known buffer limit, we introduce a new primative **read_mod** in **elim_local**. Suppose there are n equations eqn_{S_i}, $i = 1, ..., n$ in node S. An algorithmic description of **elim_local** is given as follows.

elim_local(S)
  for $i = 1, ..., n$, do
    if $mod(i,m) = 0$, then
      for $j = 1, ..., k$ do

read_mod
end j loop
endif
elim_eqn (eqn_{S_i})
end i loop

Two parameters m and k are used here. The procedure **elim_eqn**($eqnE$) has processor P participating in eliminating $eqnE$. The procedure **read_mod** involves a non-blocking read from each of the other processors, i.e., processor P tests for each processor Q whether there has been a pivot equation sent from Q which is ready for P to use. If so, P reads the message from the communication buffer and performs the associated modifications. For each cycle of an interval of m equations in S the procedure **read_mod** is executed k times. The smaller m is, the fewer read interrupts P has. When $m = 1$, P interrupts for **read_mod** before processing each equation in S. When $m = n + 1$, P never interrupts and the algorithm thus reduces to the complete pipelining version. On the other hand, the larger k is, the longer P delays at each interruption in order to clear the communication buffer some before processing the next equation.

This approach has been tested in the NCUBE and it does help increase the size of problems solvable but not very much. For example, using the complete pipelining algorithm ($k = 0$) to solve a PDE problem on a 37×37 grid made the NCUBE 1 system crash due to a communication buffer overwrote which overwrote the operating system (there was no system protection from this overflow). With $(m,k) = (1,1)$, we could solve problems up to a size 41×41 grid. On the other hand, the grid-based subtree-subcube assignment could solve larger problems than the standard subtree-subcube did. We also tested it on the NCUBE 2 which has larger buffer size than the NCUBE 1. We could only increase the solvable problem size up to a 49×49 grid before the same trouble occurred. At any rate, for a given buffer size there is a limit, theoretically, to the size of problems which can be solved for any given algorithm. This limitation is very serious for direct solvers, in our experience, on currently available machines because direct solvers involve much more communication than iterative methods. Efficiency is reduced in other ways by all the techniques we know to reduce this limitation, for example, splitting long messages into several small ones, increasing the frequency of interruptions, and synchronization delays.

The above factorization algorithm is basically a fan-out organization. Our experiments show that it is still rather inefficient even though the communication requirement has been improved by our new assignment and indexing strategies. We find that there

are two other big factors affecting its performance. At first, it is more expensive in data structure manipulation than the symmetric matrix fan-in scheme. The other factor is the extra cost in managing the column symbolic structure due to the lack of symmetry. We also find that it is inappropriate to use a uniform organization or data structure throughout the computation, especially for parallel computation, because the problem structure varies widely during the process of Gauss elimination. Based on these considerations, a new efficient organization of Gauss elimination is proposed in [Mu, Rice, 1990b] which uses the fan-in organization in portions of the process as much as possible and also employs different data structures at different stages. We illustrate its basic idea as follows. This new algorithm is implemented as the module NEW GAUSS ELIMINATION.

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

$$A\mathbf{x} = \mathbf{f} \tag{6.1a}$$

with

$$A = \begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_p & B_p \\ C_1 & C_2 & \cdots & C_p & D \end{bmatrix}, \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \\ \mathbf{x}_d \end{bmatrix}, \mathbf{f} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \\ \mathbf{f}_d \end{bmatrix} \tag{6.1b}$$

The matrices C_i and D contain all the elements of levels 1 to L. We first perform Gauss elimination on the subdomain equations $A_i\mathbf{x}_i + B_i\mathbf{x}_d = \mathbf{f}_i$ to get

$$\begin{cases} U_i\mathbf{x}_i + \tilde{B}_i\mathbf{x}_d = \tilde{\mathbf{f}}_i, \quad i = 1, \ldots, p \\ \sum_{i=1}^{p} C_i\mathbf{x}_i + D\mathbf{x}_d = \mathbf{f}_d \end{cases} \tag{6.2a}$$

where

$$A_i = L_iU_i, \quad i = 1, \ldots, p \tag{6.2b}$$

are the standard LU factorizations and

$$\tilde{B}_i = L_i^{-1}B_i, \quad \tilde{\mathbf{f}}_i = L_i^{-1}\mathbf{f}_i, \quad i = 1, \ldots, p \tag{6.2c}$$

The last set of equations is unchanged.

The next step is to eliminate the subdomain unknowns $[\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_p]^T$ from the interface equations. From (6.2a), we have

$$\mathbf{x}_i = U_i^{-1}\mathbf{f}_i - U_i^{-1}\tilde{B}_i\mathbf{x}_d \qquad i = 1, ..., p \tag{6.3a}$$

and

$$\sum_{i=1}^{p} C_i(U_i^{-1}\tilde{\mathbf{f}}_i - U_i^{-1}\tilde{B}_i\mathbf{x}_d) + D\mathbf{x}_d = \mathbf{f}_d \tag{6.3b}$$

So, (6.1a) is transformed to

$$\begin{cases} U_i\mathbf{x}_i + \tilde{B}_i\mathbf{x}_d = \tilde{\mathbf{f}}_i \qquad i = 1, ..., p \\ \tilde{D}\mathbf{x}_d = \tilde{\mathbf{f}}_d \end{cases} \tag{6.4a}$$

where

$$\begin{cases} \tilde{D} = D - \sum_{i=1}^{p} C_i U_i^{-1}\tilde{B}_i \\ \tilde{\mathbf{f}}_d = \mathbf{f}_d - \sum_{i=1}^{p} C_i U_i^{-1}\tilde{\mathbf{f}}_i \end{cases} \tag{6.4b}$$

This leads to the following algorithm.

Algorithm: A New Organization of Sparse Gauss Elimination.

Factorization.

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

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

Solution.

- for $i = 1$ to p do

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

  end of i loop.

In the above algorithm, L_i, U_i and $\tilde{B}_i$ can be calculated in each processor by a fan-in organization because they are totally local problems, neither the lack of symmetry nor a DMMP architecture affects the application of the fan-in scheme to this part. Secondly, unlike in the standard approaches, the C part of the matrix is never touched here. This allows one to avoid monitoring the dynamic change of the symbolic structure of the C part. The communication between subdomains and interfaces can thus be determined statically. This also allows one to use a dense data structure for the D part since its size is relatively small, which, in turn, makes the latter stages more efficient. Finally, we apply the above fan-out parallel sparse solver to the factorization $\tilde{D} = L_d U_d$ on the small interface submatrix. For detailed discussions of this new scheme, see [Mu, Rice, 1990b].

VI. b Substitution

Back substitution on triangular linear system solution is difficult to speed up substantially even for dense matrices [Eisenstat, Heath, Henkel and Romine, 1988], [Li, Coleman, 1988]. For sparse matrices, reasonable speed up is even more difficult to achieve. For example, [Ribbens, 1990] reports the results of an intense effort to speed up the back solve for a PDE problem using ordinary finite differences and about 16 thousand equations. With from 4 to 16 processors, the maximum speed up achieved is just over 2. This is a dismal result compared to what can be achieved in the factorization phase (or with iterative methods) in solving linear systems.

There are two phenomena here. First, the order of the back solve is essentially sequential. Parallelism at the top level is available only in those blocks of the matrix where domain decomposition has partitioned the equations into independent groups. There seems to always be one large block which is not partitioned, and some of the other blocks are not very small. Second, the amount of arithmetic in back substitution

is dramatically reduced compared to factorization, but the amount of communication is not. Since communication is very slow relative to arithmetic on current machines (and we do not expect this to change soon), communication dominates the time used.

The two paradigms of fan-in and fan-out are relevant to back substitution. The fan-in approach is more communication efficient, yet not efficient enough. One can attempt [Eisenstat, Heath, Henkel and Romine, 1988] to do part of the arithmetic on the top part of the upper triangular systems concurrently with solving for unknowns on the bottom part. If the problem is large enough, this *cyclic* approach eventually pays off, but only for very large problems. The reason is that there is not enough arithmetic to do in advance to compensate for the communication cost. Even if the communication is completely overlapped with other work (and thus can be considered to be free), the small amount of arithmetic to be done provides little benefit for modest sized systems.

Observe that most efficient methods for factorization leave the triangular system poorly distributed among the processors for efficient back substitution. Almost every time a new unknown is to be found, the computation moves to another processor. The idea of *copied blocks* [Mu and Rice, 1990c] remedies this at the cost of using additional storage. The technique is to observe that in the factorization phase the pivot equations are passed to processors for elimination purposes. These can be selectively saved so that, at the end of the elimination, processor k has a copy of rows $(k-1)N/p+1$ to kN/p for N equations and p processors. The back substitution is then carried out on the copy of the triangular system and there are only p communication steps required rather than the nearly N required if the triangular system is solved in its ordinary locations. [Mu and Rice, 1990c] show that the compute ahead approach can be used with some profit. It is, of course, difficult to evaluate the trade off between memory and time; there are cases where each is the critical resource.

VII. IMPLEMENTATION

We briefly describe in this section the implementation of the geometric approach as part of the current prototype of the Parallel ELLPACK system [Houstis, et al., 1989]. The system structure consists of five phases as illustrated in Figure 7.1.

(1) Domain Decomposition

This phase is composed of two steps. At first, the ELLPACK domain processor [Rice, Boisvert, 1985] discretizes a domain Ω to a **domain-decomposition-grid** Ω_d using a tensor-product grid to numerically represent the geometry. Then we provide a

domain decomposition module ONE-WAY-NESTED-DISSECTION for performing the decomposition as described in Section II. The output of this module is data for the Parallel ELLPACK *domain decomposition interface* mainly consisting of two arrays **i9subd** and **i9intf** which associate each subdomain element or interface grid point with an id of a generalized subdomain (Ω_{ij} or separator subdomain) of the domain decomposition. The id is actually the corresponding node id of the subdomain in the elimination tree. The geometric information of the domain decomposition is represented in this data structure in the same way as described in [Houstis, et al., 1989]. However, it also provides the information of the elimination tree for the geometric approach. In other words, i9subd represents the square mesh partition for Ω, while i9intf represents the further nested dissection decomposition in the separators for interfaces. An example is illustrated in Figure 7.2 for a rectangular domain Ω with a 9×9 grid Ω_d and a 2×2 square mesh is used for the domain partition.

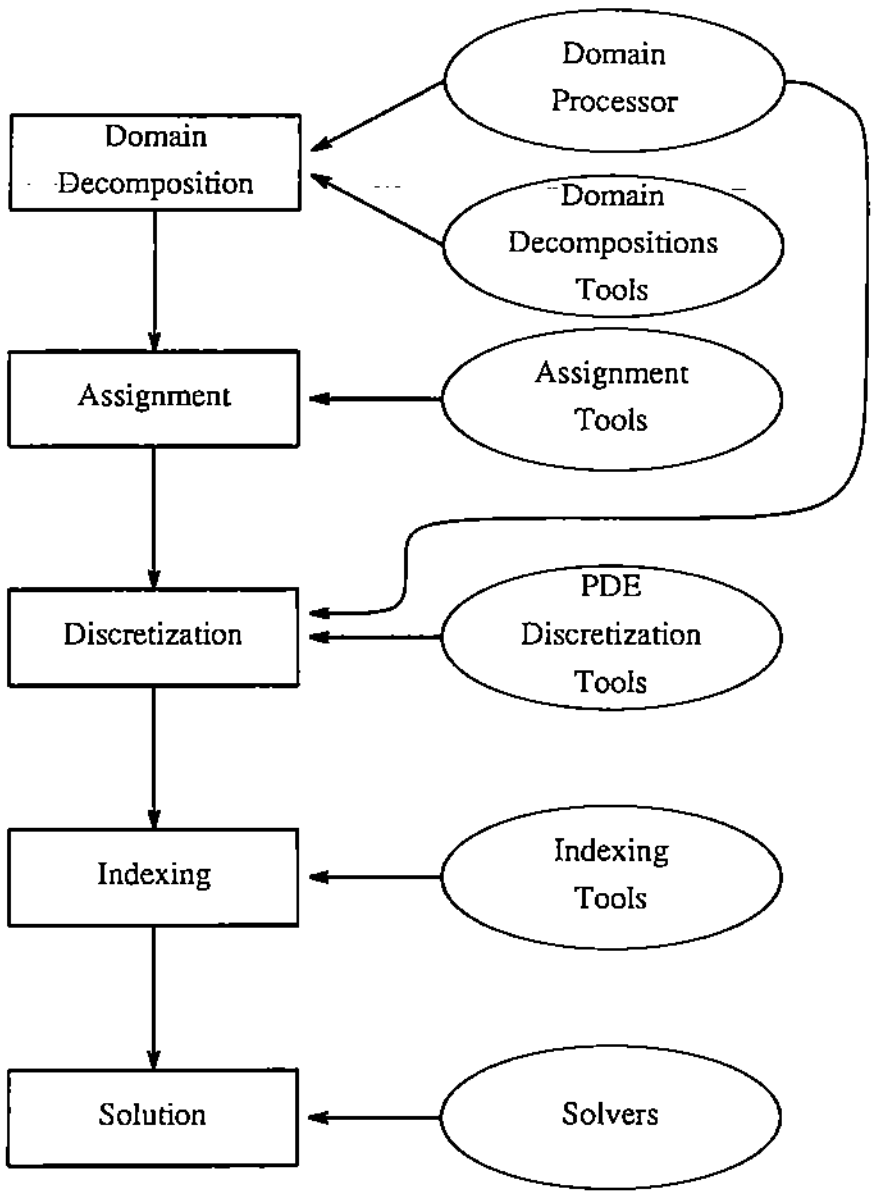
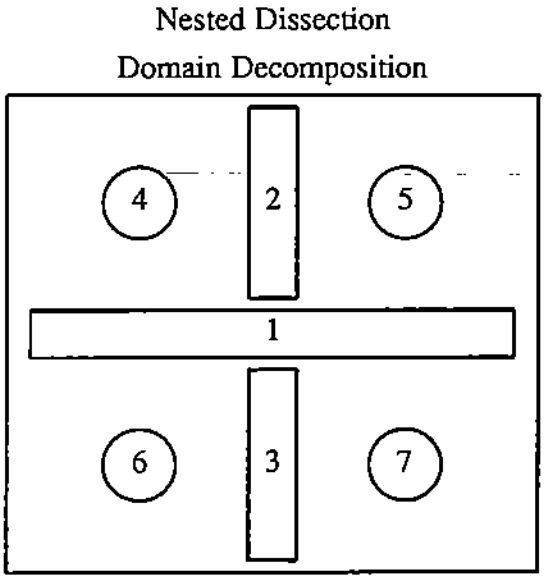
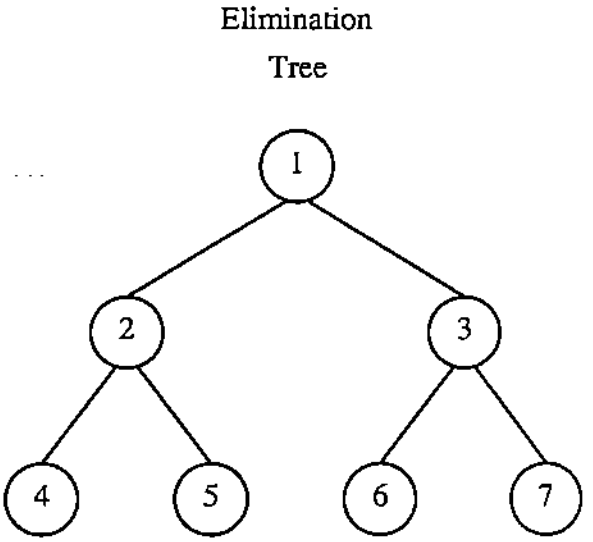


Figure 7.1. The structure of parallel ELLPACK as it relates to the geometric approach for parallel sparse matrix algorithms.





i9subd

4	4	4	4	5	5	5	5
4	4	4	4	5	5	5	5
4	4	4	4	5	5	5	5
4	4	4	4	5	5	5	5
6	6	6	6	7	7	7	7
6	6	6	6	7	7	7	7
6	6	6	6	7	7	7	7
6	6	6	6	7	7	7	7

i9intf

0	0	0	0	2	0	0	0	0
0	0	0	0	2	0	0	0	0
0	0	0	0	2	0	0	0	0
0	0	0	0	2	0	0	0	0
1	1	1	1	1	1	1	1	1
0	0	0	0	3	0	0	0	0
0	0	0	0	3	0	0	0	0
0	0	0	0	3	0	0	0	0
0	0	0	0	3	0	0	0	0

Figure 7.2. A domain decomposition interface example.

(2) Assignment

An assignment module assigns each grid point in Ω_d to a processor, which sets the output *assignment interface* mainly consisting of array **i9assn**. Two assignment modules SUBTREE-SUBCUBE and GRID-SUBTREE-SUBCUBE have been implemented for standard subtree-subcube and grid-based-subtree-subcube, respectively. There are two steps in these modules. At first, the real hypercube architecture is

mapped to a virtual mesh architecture by the gray code. Then grid points are mapped to the virtual mesh according to their geometric positions located from the *domain decomposition interface.* An array **i9leaf** is also set which gives the corresponding leaf node id in the elimination tree (the starting point of the elimination path to the root node) for each processor. Both *decomposition* and *assignment* are executed on the host processor.

(3) Discretization

The discretization phase potentially consists of two steps. The first step is the geometric discretization. Local grids are set by the domain processor for all subdomains (or tree nodes) of the domains decomposition. The union of these local grids forms the **solution grid** Ω_s which is used to discretize the PDE problem and the relation between Ω_s and Ω_d is also established. In general, Ω_s is different from Ω_d and usually much finer than Ω_d. In many applications, such as multigrid methods, adaptive methods by grid refinement (or *h*-version finite element methods), there are even a sequence of solution grids Ω_{s1}, Ω_{s2}, ... with respect to one decomposition grid Ω_d.

The second step is the PDE discretization. Local discrete equations are formed based on the local grids and the correspondence between unknowns and grid points in Ω_s are established. Notice that an unknown is not necessarily a function value at a grid point as in the case of COLLOCATION methods.

The assignment of unknowns/equations to processors is implied in the following way. For each unknown/equation, we first locate its related local grid point **x** in Ω_s then from the relation between Ω_s and Ω_d, a corresponding grid point **X** in Ω_d found. Finally, the processor, with $id = i9assn(\mathbf{X})$ from the *assignment interface*, is assigned to this unknown/equation. The discretization computation is distributed among the processors.

(4) Indexing

This phase sets the standard *ELLPACK indexing interface.* For the sake of representing the elimination tree structure, two arrays **i9fst** and **i9lst** are also set which give the indices of the first and last, respectively, unknown/equation for each tree node. A module DISSECTION-MINIMUM DEGREE-FRONTAL for the indexing as described in Section V has been implemented. The *indexing* is done globally by the host processor.

(5) Solution

The solution phase solves the linear system of algebraic equations. Two solution modules, PARALLEL SPARSE and NEW GAUSS ELIMINATION, have been implemented for solving general nonsymmetric systems. The factorization algorithms used are described in Section VI and the back substitution just uses a standard algorithm because no efficient parallel sparse triangular solvers are available so far. Therefore, we only measure the performance of the factorization part in the data below. A dynamic data structure is used for PARALLEL SPARSE as described [Mu, Rice, 1990a]. The mixed data structure used for NEW GAUSS ELIMINATION is presented in [Mu, Rice, 1990b]. The standard data structure in *Parallel ELLPACK* interfaces is converted to the local data structures by the corresponding driver routines in these solution modules.

VIII. PERFORMANCE EXPERIMENTS

We present experimental performance data on both the first and the second generations of the NCUBE hypercube machines to show the effects and importance of blending algorithm components in solving PDE problems. Our model problem is a *Poisson* equation with *Dirichlet* boundary condition on a rectangular domain using the geometric approach with *five-point-star* discretization. Even though the problem is actually symmetric, we still treat it as a non-symmetric one since we want to examine the effects of algorithms for general problems.

It is not our intention here to present an exhaustive, or even nearly so, set of performance data on algorithms that can be created using the various components. Our intention is to show that the component choices have major effects on performance (which is no surprise) and that they interact in strong ways with each other and the hardware characteristics. Our thesis is that *there is probably no universally best choice for any of the algorithm components*. If this thesis is correct, it is a discouraging conclusion as it implies that achieving very high performance requires the continual creation of sparse matrix algorithms which exploit the special properties of the PDE problem and the hardware/software environment.

Our approach is to carry out experiments when most of the algorithm components are held fixed. We then vary a few components, often just one, observe the results and suggest conclusions about performance in general. We are also able to make comparisons with some performance data published by others involving particular algorithm components. Our experiments were on the NCUBE 1 and NCUBE 2 machines, usually using 16 processors. We do not discuss their architecture here, see [Palmer, 1986], but note that they have considerably different performance parameters and yet both have the

low communication to computation speed ratios typical of hypercubes and distributed memory machines in general.

Throughout these experiments we have the following four algorithm components fixed.

Domain decomposition: square mesh

Interface organization: alternating one way nested dissection hierarchy

Matrix organization: by rows

Discretization: ordinary finite differences

We first compare the performance of two assignments, the standard subtree-subcube and grid-based subtree-subcube, in Section IV. The module combinations are listed in Table 8.1.

Table 8.1. Module combinations for testing assignment components.

Assignment	SUBTREE-SUBCUBE (*standard*)	GRID-SUBTREE-SUBCUBE (*grid*)
Indexing	DISSECTION-MINIMUM DEGREE-FRONTAL (wrapping in each seperator)	
Solution	PARALLEL SPARSE	

Figure 8.1 shows the number of messages versus $N^{1/2}$ for both *standard* and *grid*. The total message volume versus N is shown in Figure 8.2. We can observe a substantial reduction in communication requirements using the *grid* assignment strategy.

Figures 8.3 and 8.4 then show the individual timing curves of the 16 processors for the *grid* and *standard* assignments, respectively. We see that the former has a worse load balance than the latter even though the former has more efficient communication. The reason for the worse load balance, as discussed in Section V, is that we do not couple the *grid* assignment with a compatible local indexing component. The one used, in each separator, favors the communicationally inefficient *standard* assignment. Therefore, the overall algorithm performance still has not been improved very much.

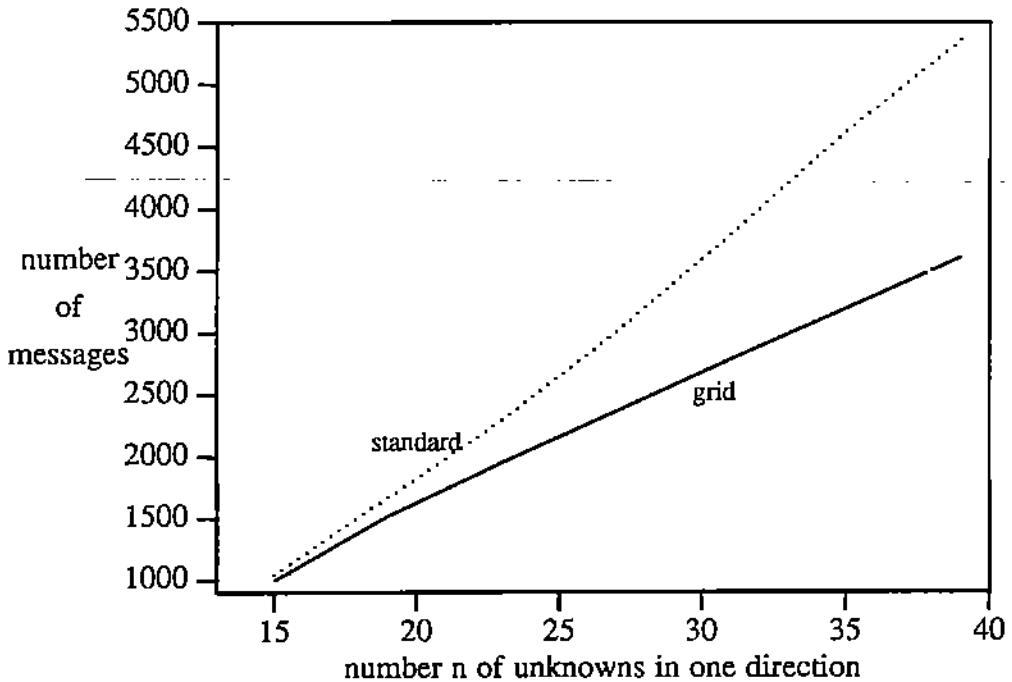


Figure 8.1. Total number of messages S vs. number $n = N^{1/2}$ of unknowns in one direction. There were 16 processors used.

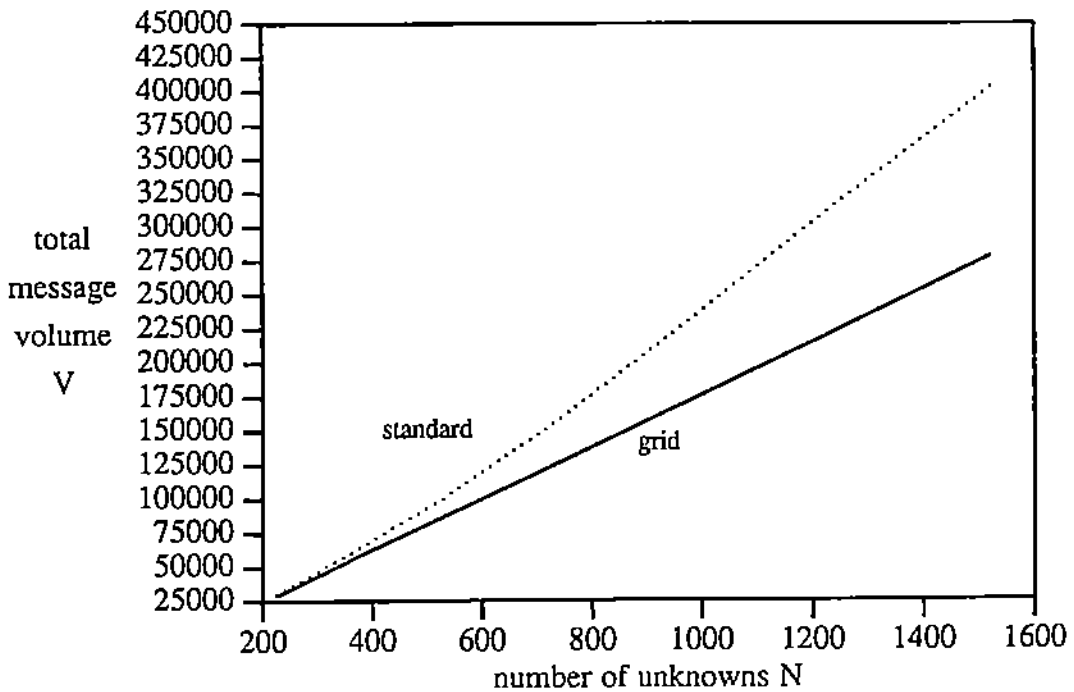


Figure 8.2. Total message volume V vs. number N of unknowns. There were 16 processors used.

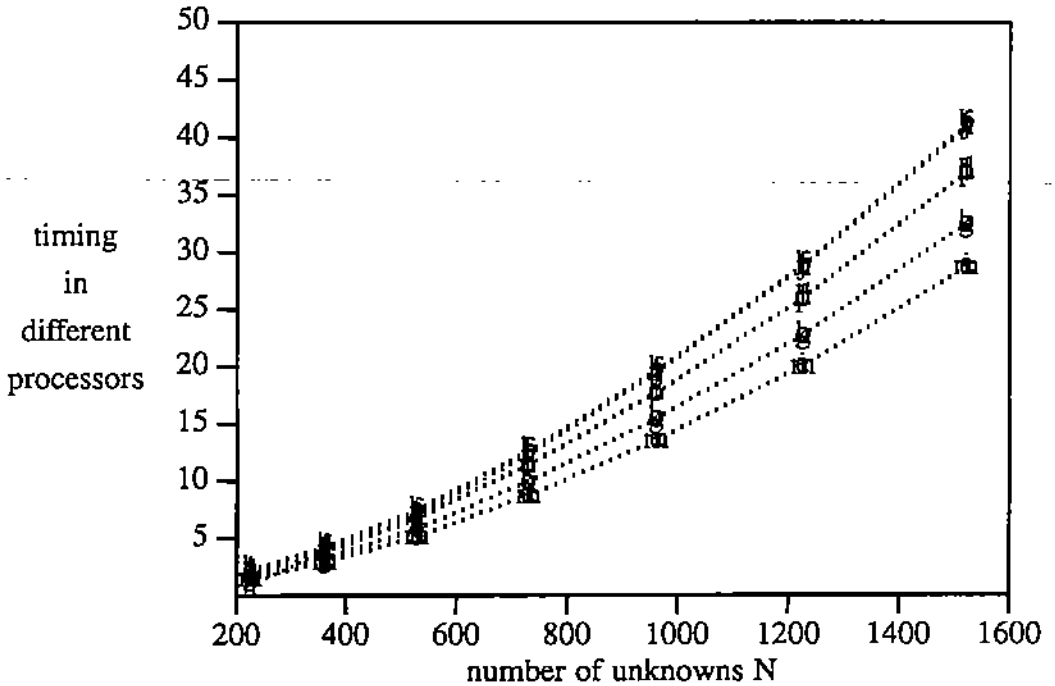


Figure 8.3. Timing vs. number N of unknowns for the *grid* assignment. The NCUBE 1 is used with 16 processors.

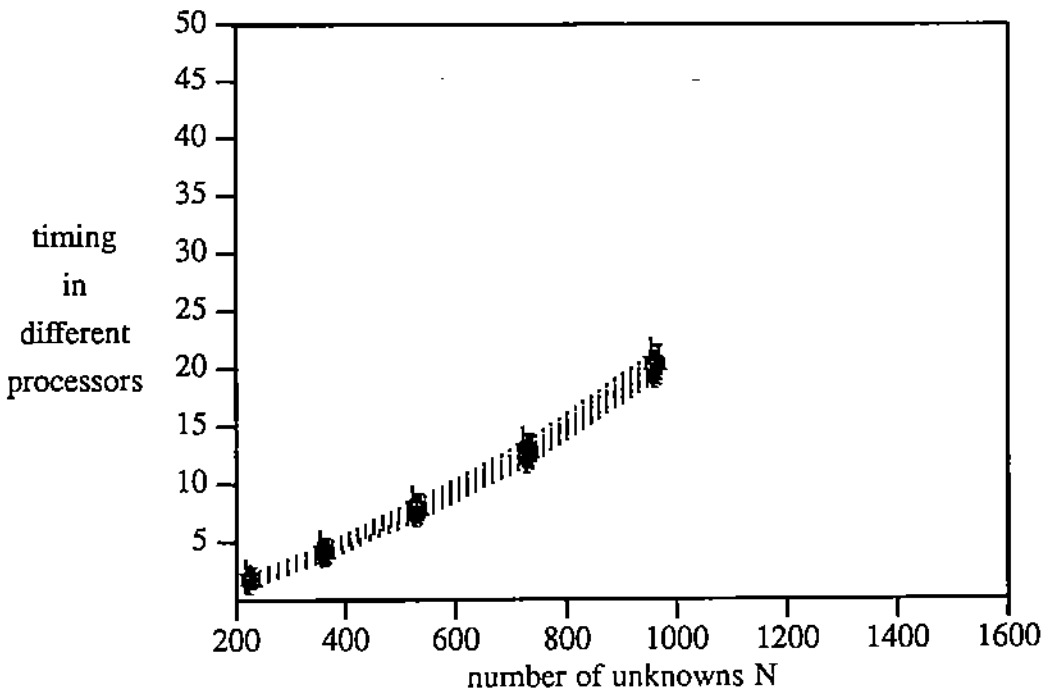


Figure 8.4. Timing vs. number N of unknowns for the *standard* assignment. The NCUBE 1 is used with 16 processors.

We next experimentally examine how the local indexing affects the load balance and the overall performance. Four local indexings as described in Table 8.2 are tested in the module DISSECTION-MINIMUM-DEGREE-FRONTAL. The segmentwise-wrapping for a separator in Table 8.2 is illustrated in Figure 5.2.

Table 8.2 Local indexing components used in the second experiment. The *level* refers to level in the block elimination tree.

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

The assignment and solution modules used are GRID-SUBTREE-SUBCUBE and PARALLEL SPARSE, respectively, and the test problem is a 37×37 grid with 1225 unknowns using 16 processors on the NCUBE 1. The corresponding timing data are listed in Table 8.3.

From Table 8.3 we draw the following conclusions. The local indexing has substantial effects on the load balance and the overall efficiency. But, these effects are hard to predict. Sometimes, such as for type 2, one gets good load balance, but a worse overall performance because keeping every processor equally busy does not imply overall efficiency. This indexing might introduce more synchronization and so on. None of the tested local indexings behave satisfactorily in both load balance and overall performance. Some adaptive strategies might help to achieve better results. In addition, the load balance effects above only relate to the computation ordering; one could also distribute the work differently. For example, the processors assigned to those corner or boundary subdomains obviously have less work to do. Similar situations occur if a PDE presents singularities and an adaptive grid is used. [Zmijiwski, 1989] suggests to use unbalanced separators to balance the work load. If this is done then the local indexing should also take this factor into account. We believe that this is a moderate complication in the algorithms but worthwhile for achieving very high performance.

Table 8.3 Timing data for different local indexings. The maximum and minimum are over the sixteen processors of the NCUBE 1.

Interface Indexing Type	Maximum timing	Minimum timing
0	28.86	19.88
1	28.37	27.53
2	29.20	28.36
3	28.05	19.40

We next study some effects of how parallelism is exploited in the solution component. We fix the assignment component to be GRID-SUBTREE-SUBCUBE, the indexing to be DISSECTION-MINIMUM DEGREE-FRONTAL with the type 2 local indexing.

We explore some of the trade-offs among parallelism, data structure, sparsity, communication pattern, algorithm organization and so on in the overall performance of the nested dissection domain decomposition. For a 4×4 domain decomposition, as the example of Section V, there are five levels in the elimination tree. We test five cases, *case* ℓ, $\ell = 1, \ldots, 5$, where nodes on the top ℓ levels are grouped and treated as one dense system, or a generalized node, to which **elim_local** is applied. The two extreme cases $\ell = 1$, and 5, respectively, correspond to the full dissection decomposition and no decomposition at all. The case $\ell = 2$ is where the standard domain decomposition (or substructure) is applied and no further partition is used to the interface equations. Table 8.4 lists the maximum timing data for this experiment. It shows that the efficiency of parallelism in using the sparsity represented in the elimination tree decreases as the level moves from bottom to top.

Table 8.4 Maximum execution time in seconds using sparsity and parallelism to different extents within the block elimination tree.

Case l =	1	2	3	4	5
Maximum time	29.20	35.18	39.77	41.33	63.49

We next study how the lack of symmetry seriously degrades the performance of parallel sparse matrix Gauss elimination. As mentioned earlier, the algorithm of PARALLEL SPARSE monitors the column symbolic structure (C-INFO) which is used to generate the destination lists for communicating pivot equations (multicast tasks). This requires both manipulating the C-INFO data structure and extra communication. Our experiment is as follows. For the last (top) node of the elimination tree, we consider in **elim_local** that the multicast tasks are very close to broadcast tasks because the problem at this stage is almost dense. Therefore, the multicast is replaced by the broadcast and involving the C-INFO data structure is thus avoided. This approach cannot be used for all nodes since it introduces too much synchronization for sparse matrices as seen in Table 8.4. If one merely replaces the multicast by directly sending a message to all other processors (not a broadcast since no syncronization), then it will not only heavily increase the communication cost, but it also causes communication buffer overflow problems. Because our test problem is actually symmetric, we can, for all other tree nodes, make use of the information in the corresponding row instead of using C-INFO. This is what symmetric (Cholesky) sparse matrix solvers do. This change makes the execution time drop dramatically from 29.20 seconds to 6.41 seconds.

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

Finally, we list our experiment data on the NCUBE 2 for the NEW-GAUSS-ELIMINATION module which uses different algorithm organizations and data structures at different stages.

Table 8.5 Parallel speed up performance of NEW-GAUSS-ELIMINATION on the NCUBE 2.

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

We observe a substantial improvement in the parallel speed up of NEW-GAUSS-ELIMINATION over PARALEL SPARSE.

We also present in Table 8.6 the speed ups reported in [Ashcraft, Eisenstert and Liu, 1990] using 16 processors for similar PDE problems.

Table 8.6 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	125×125 grid, nine-point-star	7503
fan-out	5.54	2614 unknowns	2614
multifrontal	9.50	65×65 grid, nine-point-star	3969

All the algorithms in Table 8.6 are for Cholesky factorization, so no nonsymmetric difficulties are present. From Table 8.6 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 NEW-GAUSS-ELIMINATION.

IX. CONCLUSIONS

We have studied various algorithmic components and performance aspects of PDE sparse solvers. We observe that compatible coupling of the components of PDE solvers is very crutical in achieving a satisfactory overall performance. The geometric approach is natural and flexible for constructing efficient PDE solvers by blending algorithmic components with each other; selecting appropriate assignments, orderings, data structures, organizations, communication protocols and so on. It is also effective for tailoring the solver to problem properties, such as geometry, physics, symmetry, sparsity, architecture etc. For general PDE applications, nested dissection is only effective on the global domain decomposition level and in ordering subdomain interfaces. For local subproblems, such as in one subdomain, an interface separator or a supernode, it is better to insulate the interior effects from the outside as long as possible. Sparse matrix techniques should be used in different ways at different elimination stages because the problem's nature varies from very sparse to dense during the solution process. In one word, there is not an optimal choice for any one of algorithmic components because of their nutual interactions and of the effect of application properties.

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. Ashcraft, C., S.C. Eisenstat, J.W.H. Liu, B.W. Peyton and A.H. Serman (1990), "A Compute-Ahead Implementation of the Fan-in Sparse Distributed Factorization Scheme", ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN.
3. Chan, Tony (1988), "The Physics of the Parallel Machines", CAM Report 88-38, Dept. of math., UCLA, Los Angeles, CA 90024.
4. Chrisochoides, N.P., C.E. Houstis, E.N. Houstis, P.N. Papachiou, S.K. Kortesis and J.R. Rice (1990), "DOMAIN DECOMPOSER: A Software Tool for Mapping PDE Computations to Parallel Architectures", *Proceedings of Fourth Conference on Domain Decomposition Methods*, Moscow, May 1990.
5. Duff, I.S. (1986), "Parallel Implementation of Multifrontal Schemes", *Parallel Computing*, 3, pp. 193-204.

6. 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.

7. Forsythe, G.E. and W.R. Wasow (1960), *Finite Difference Methods for Partial Differential Equations*, John Wiley and Sons, Inc.

8. Geist, G.A. and M.T. Heath (1986), "Matrix Factorization on a Hypercube Multiprocessor", *Hypercube Multiprocessors*, (M.T. Heath, Ed.), SIAM Publications, pp. 161-180.

9. George, A., M. Heath, J. Liu, and E. Ng (1988), "Sparse Cholesky Factorization on a Local-Memory Multiprocessor", *SIAM Sci. Stat. Comput.*, Vol. 9, no. 2, 327-340.

10. 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.

11. George, A., J. Liu, and E. Ng, (1988), "A Data Structure for Sparse QR and LU Factors", *SIAM J. Sci. Stat. Comput.*, **9**, pp. 100-121.

12. George, A., and E. Ng (1988), "Parallel Sparse Gaussian Elimination with Partial Pivoting", Tech. Rept., ORNL/TM-10866, Oak Ridge National Laboratory, Oak Ridge, TN.

13. Houstis, E.N., W.F. Mitchell and J.R. Rice (1985), "Collocation Software for Second Order Elliptic Partial Differential Equations", *ACM Trans. Math. Software*, **11**, 379-412.

14. Houstis, E.N., et al., (1989), "Parallel ELLPACK PDE Solving System", Computer Sci. Dept., Purdue University, Tech. Rept. CSD-TR-912, CAPO Rept. CER-89-20.

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

16. Li, G. and T.F. Coleman (1988), "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor", *SIAM J. Sci. Stat. Comput.*, Vol. 9, 485-502.

17. Liu, J. (1988), "Equivalent Sparse Matrix Reordering by Elimination Tree Rotations", *SIAM J. Sci. Stat. Comput.*, Vol. 9, no. 3, 424-444.

18. Liu, J.W.H. (1989), "The Minimum Degree Ordering with Constraints", *SIAM J. Sci. Stat. Comput.*, Vol. 10, No. 6, pp. 1136-1145.

19. Liu, J.W.H., E. Ng and B.W. Peyton (1990), "On Finding Supernodes for Sparse Matrix Computations", ORNL/TM-11563, Oak Ridge National Laboratory, Oak Ridge, TN.

20. 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.

21. 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.

22. 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.

23. Mu, M. and J.R. Rice (1990b), "A New Organization of Sparse Gauss Elimination for Solving PDEs", Computer Sci. Dept., Purdue Univ., Tech. Rept. CSD-TR-991, CAPO Rept. CER-90-22.

24. Mu, M. and J.R. Rice (1990c), "A Copied Block Approach for Back Solve of Sparse Triangular System on Distributed Memory Multiprocessors". In preparation.

25. Palmer, J.F. (1986), "A VLSI Parallel Supercomputer", *Hypercube Multiprocessors*, (M.T. Heath, Ed.), SIAM Publications, pp 19-26.

26. Ribbens, C. (1990), "On Parallel ELLPACK for Shared Memory Machines", Tech Rept. 90-46, Computer Science Department, VPI and SU.

27. Rice, J. (1986), "Parallel Methods for PDEs", in *The Characteristics of Parallel Algorithms* (Jamieson, Gannon, Douglass, Eds.) Chapter 8, MIT Press, 209-231.

28. Rice, J.R. (1989), "Doing Linear Algebra Blind Across Boundaries", Computer Sci. Dept., Purdue Univeersity, West Lafayette, IN.

29. Rice, J.R. and R.F. Boisvert (1985), *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York.

30. Rothberg, E. and A. Gupta (1990), "Efficient Sparse Matrix Factorization on High-Performance Workstations – Exploiting the Memory Hierarchy", Dept. of Computer Science, Stanford University.

31. Strang, G. and G.J. Fix (1973), *An Analysis of the Finite Element Method*, Prentice-Hall, Englewood Cliffs, NJ.

32. Zmijewski, E. (1989), "Limiting Communication in Parallel Sparse Cholesky Factorization", TRCS89-18, Department of Computer Science, University of California, Santa Barbara, CA.