PARALLEL NUMERICAL ALGORITHMS AND SOFTWARE

It has been recognized that a large number of scientific problems have led to models whose simulation necessitates the solution of algebraic equations. Moreover, the cost of setting up and solving these systems dominates the overall complexity of the simulation. In practice, the overall quantitative and qualitative accuracy that these computational partial differential equation (PDE) models can achieve in representing the physical situations or artifacts depends to a great extent upon the computer resources available to solve the corresponding PDE discrete algebraic systems. The recent advances in high-performance computing technologies have provided an opportunity to significantly speed up these computational PDE models and dramatically increase their numerical resolution and complexity.

The purpose of this article is twofold. First, we review the various parallelization techniques proposed to speed up the existing computational PDE models, which are based on the *divide and conquer* computational paradigm and involve some form of decomposition of the geometric or algebraic data structures associated with these computations. Second, we review the parallel algorithms proposed to solve various classes of algebraic systems which are applicable to discrete PDE systems. For the sake of brevity of this exposition we focus on computational models derived from elliptic PDE models. Most of the parallelization techniques presented here are applicable to general semidiscrete and steady-state models.

Specifically, we consider PDE models consisting of a PDE equation $Lu = f$, defined on some region Ω and subject to some auxiliary condition $Bu = g$ on the boundary of Ω ($= \partial\Omega$). It appears that one can formulate many (thousands) computational models to simulate the above general mathematical model. They depend on the approximation technique selected to discretize the domain of definition, the specific PDE and boundary conditions, and so on. In this article, we have selected to review parallel computational models based on the most popular discretization techniques, such as finite-difference approximations of L and B and finite-element approximations of the solution u. In these parallel computational models, the continuous PDE problem is reduced to a distributed *sparse system* of linear equations. As the operator, boundary conditions, and domain range from the most general (e.g., nonseparable, non-self-adjoint) to the simplest (Poisson, rectangular domain), the corresponding finite-difference or finite-element system of equations can be solved by general sparse iterative solvers or direct "rapid" solvers. In this article we discuss the various proposed parallel linear algebraic equation solvers and the grid/mesh partitioning strategies for the implementation of these two classes of PDE solvers on a virtual parallel machine environment.

This article is organized as follows. We first focus on the parallelization techniques of general elliptic PDE computations that both allow the reuse of existing ("legacy") elliptic PDE software parts and provide a *template* or *framework* to build new parallel elliptic PDE software. We then review the needed infrastructure to support these methodologies. A list of available parallel PDE problem-solving environments is also included. The next section deals with parallel dense linear algebraic solvers. We first present a review of parallelizing techniques for the *LU* factorization method and then we present a unified view of the parallel rapid elliptic PDE solvers using tensor product formulation. In the final section we discuss parallel sparse linear algebraic solvers that are already available in a form of software. Both direct and iterative approaches are presented.

PARALLEL ELLIPTIC PDE SOLVERS

The plethora of numerical elliptic PDE solvers can be distinguished and classified by the levels of grid(s)/mesh(es) used to approximate the continuous PDE model (i.e., single-level or multilevel), the refinement of the grid as a function of the discretization error in an intermediate computed solution [i.e., static or dynamic (adaptive)] and the implementation structure of the PDE software (i.e., multisegment or single-segment). In this article we have selected to review the parallelization techniques proposed for single-level grid elliptic PDE solvers for general and model elliptic PDE boundary value problems. Some of the parallelization approaches presented here are applicable to multilevel elliptic PDE solvers (see Ref. 1). The parallelization of adaptive elliptic PDE solvers is a much harder problem (2).

Parallelization Methodologies for "Legacy" PDE Software

There is significant "legacy" software for elliptic and parabolic PDEs. It represents hundreds of labor-years of effort which will be unrealistic to expect to be transformed by "hand" (in the absence of parallelizing compilers) on some virtual or physical parallel environment. The legacy software can be classified into two large classes. The first class contains customized PDE software for specific applications. An example of such software is PISCES (3), which is usually difficult to adapt to the simulation of a different application. The second class contains PDE software that supports the numerical solution of well-defined mathematical models which can be used easily to support the simulation of multiple applications. The first class tends to be application-domain-specific, and thus the parallelization efforts and results appear in many diverse sources. In this article we review the parallelization techniques proposed for the second class of PDE software. Some of the public domain "legacy" software available in the //ELLPACK system (4) are the following: ELLPACK, FIDISOL, VECFEM, CADSOL, PDEONE, PDECOL, PDETWO, MGGHAT.

It is worth reminding the reader that the majority of the code of each PDE system is implementing the geometric and the PDE model discretization phases. This tends to be the most knowledge-intensive part of the code. The rest of the code deals with the solution of the discrete finite-difference or finite-element equations, which is better understood, and many alternative solution paths exist. We review those efforts that have already been implemented in the form of software. In Table 1 we summarize the above observations, and in its last column we estimate the parallelization effort needed to convert or reimplement the components of the legacy PDE code into some parallel environment "by hand". It is clear that any parallel methodology that attempts to reuse the PDE discretization software parts is well-justified. We describe below three parallel methodologies that are based on some "optimal" partitioning of the discrete PDE geometric data structures (i.e., grids and meshes). Figure 1 depicts these three decomposition approaches for a two dimensional region and message passing computational paradigm. The two left most paths in Fig. 1 depict methodologies that support the reuse requirement. The third path provides a framework to develop new customized parallel code for the discretization part of the PDE computation. All three approaches assume the availability of parallel linear solvers implemented on distributed algebraic data structures obtained through some "optimal" partitioning of the corresponding PDE geometric data structures. Next, we elaborate on these approaches and indicate the required infrastructure.

The left path in Fig. 1 depicts an off-line parallelization approach, referred to as M^+, which assumes that the discretization of the PDE model is realized by an existing sequential "legacy" PDE code, while it goes off-line to a parallel machine to solve the system of discrete equations. For the parallel solution of the discrete PDE equations, a decomposition of the sequentially produced algebraic system is required. It can be either *implicitly* obtained through a decomposition of the mesh or grid data or *explicitly* specified by the user. Then, the partitioning system is downloaded on the parallel machine. This is the most widely used methodology, since it allows for the preservation of the most knowledge-intensive part of the

Table 1. The Complexity of the Elliptic PDE Software Parts and Estimates of the Parallelization Effort Needed

Components	Computational Intensity	Knowledge Intensity	Parallelization Effort Needed
Geometric discretization	$O(N)$	Very high	Significant
PDE model discretization	$O(N)$	Very high	Significant
Solution	$O(N^{\alpha})$, $1 < \alpha \leq 3$	Well understood–high	Relatively easy
Graphical display of solution	$O(N)$	High	Needs specialized hardware

N denotes the size of the discrete problem.

code and for speeding up the most computationally intensive one. The obvious disadvantage of this approach is the memory bottleneck of the sequential server. To address this problem, various off-line pipeline techniques have been proposed. The current version of the //ELLPACK system includes a software tool to support this methodology for a large class of legacy software systems available. The tool is self-contained and can be used for any PDE software and virtual parallel machines supported by standards such as MPI. The input to this tool consists of the system and a partitioning of the associated matrix. The partitioning of the matrix problem can be obtained either explicitly by decomposing the matrix graph or implicitly by decomposing the discrete geometric data (i.e., mesh or grid). A comprehensive overview of the explicit matrix partitioning techniques and their performance evaluation can be found in Ref. 5.

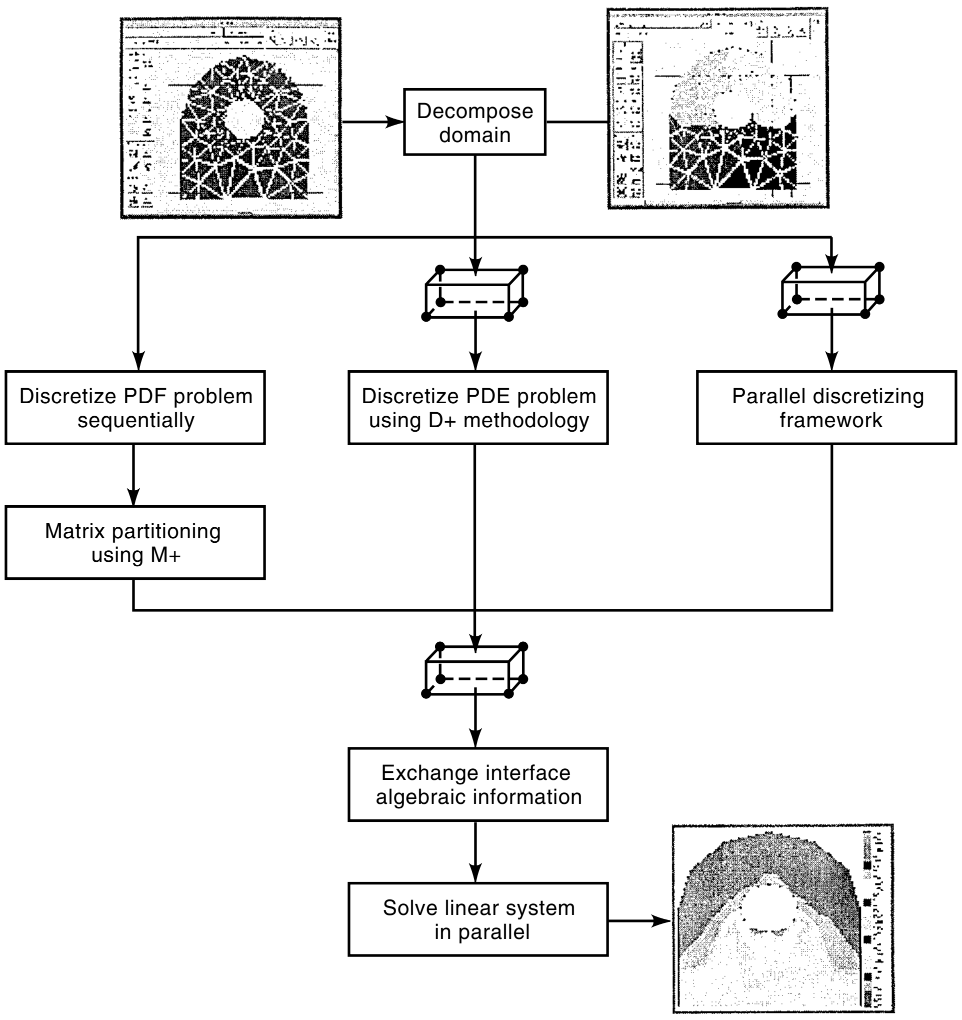
Decompose domain
Discretize PDF problem sequentially
Discretize PDE problem using D+ methodology
Parallel discretizing framework
Matrix partitioning using M+
Exchange interface algebraic information
Solve linear system in parallel

Figure 1. Three domain-decomposition-based parallel methodologies for elliptic PDEs. The left path depicts an off-line parallel approach for solving the sequentially generated PDE equations, the center path depicts an on-line nonoverlapping domain decomposition approach capable of reusing existing discretization PDE software, and the right path depicts a framework for developing new parallel PDE software.

The right path in Fig. 1 corresponds to a framework for developing customized PDE software. It is defined by a set of predefined decomposed geometric and algebraic data structures and their interfaces. The decomposition of the PDE data structures is chosen so that the underlying computations are uniformly distributed among processors and the interface length is minimum. Later, we review the proposed geometric decompositions and the parallel algebraic solvers to support this framework. This parallel framework has been used by many researchers to implement PDE-based applications and to develop general PDE software (4). The parallel PDE solvers implemented on the above framework are distinguished primarily by the way they handle the interface equations and unknowns. An overview of the various parallel solution strategies proposed for handling the interface and interior equations can be found in Ref. 6. The simplest of these parallel strategies calls for the implementation of efficient sequential algebraic solvers on the framework data structures through the use of parallel sparse basic linear algebra subroutines (BLAS) (7) that employ message passing primitives to exchange or accumulate interface quantities and carry out matrix–vector and vector–vector operations. The advantage of this approach is the fact that no new theory is required. Such parallel PDE solvers based on certain instances of finite-difference and finite-element schemes for elliptic PDEs can be found in //ELLPACK system (4,8). These PDE solvers are described in Ref. 9 together with their performance evaluation.

The center path in Fig. 1 illustrates a third methodology for developing parallel PDE software that supports the reuse of existing PDE codes and attempts to address the shortcomings of the previous two. It is referred to as D^+. The basic idea of this approach is to use the mesh/grid decomposition to define a number of auxiliary PDE problems that can be discretized independently using the "legacy" PDE codes. Depending on the PDE operator and the approximation scheme used, appropriate continuous *interface conditions* must be selected to guarantee that the parallel on-line generated system of equations is complete and equivalent (apart from round-off error) to the sequential discrete algebraic system. In some instances a data exchange among processors might be required to complete the system of algebraic equations. A software environment that supports the D^+ approach for elliptic PDEs is available in the //ELLPACK system.

Discrete Geometric Data Partitioning Strategies and Software

The parallel methodologies considered in this article are based on some decomposition of the PDE discrete geometric data (i.e., grid or mesh). Without loss of generality, we discuss the various proposed decomposition strategies in terms of finite-element meshes. The formulation and implementation of this phase of the proposed parallel methodologies is often done at the topological graph of the finite element mesh $G = (V, E)$ of a domain Ω, where V denotes the set of elements or nodes and E is the set of edges of G that represent the connectivity of the vertices V with its neighbors. The mesh decomposition is usually defined in terms of several optimality criteria that include *load balancing* (subdomains of almost equal size), *minimum interface length* (minimum number of common edges or nodes between subdomains), *minimum subdomain connectivity* (minimum number of neighbor subdomains), *minimum bandwidth of the local matrix problem,* and *optimal aspect ratios of the subdomains* (local matrix problem well-conditioned). The problem of graph partitioning the subject solely to the first two criteria has been found to be extremely hard to achieve computationally, and therefore most of the proposed partitioning strategies are approximate (i.e., heuristic) in nature. These heuristics have been found to be very costly even for moderate-sized PDE problems (9). Two "fast" alternative strategies have been formulated and implemented in parallel for grid (9) and mesh (10), respectively, which are based on an encapsulation approach and easily outperform the ones that are based on the partition of the exact grid. Unfortunately, this approach cannot be generalized for finite-element meshes.

The heuristics that are commonly used for mesh partitioning are based on a variety of techniques and methodologies which we briefly review next.

Neighborhood Search Schemes. This class consists of heuristics that are based on some neighborhood search scheme utilizing the connectivity information of the mesh graph G. For these schemes, the partitioning of G is equivalent to the construction of a traversal tree from the mesh graph G. Two well-known such schemes are based on *depth-first* and *breadth-first* search strategies (11). If the traversal order scheme remains fixed for the entire mesh graph G, then the searching strategy is called *stripwise;* and if it is allowed to change after the formulation of each subdomain D_i, then the search is called *domainwise.* Another set of neighborhood search heuristics are the ones used for bandwidth reduction of a matrix. A well-known such ordering scheme is the so-called reverse Cuthill–McKee (RCM). Other graph-based mapping heuristics and their performance are presented in Ref. 12. Various implementations of the above heuristics for finite-element meshes and grids together with their performance evaluation are reported in Refs. 10, 13, and 14.

Spectral Search Heuristics. According to these search schemes the vertices V are visited (sorted) in the order defined by the size of the components of an eigenvector or a combination of eigenvectors of the Laplacian matrix $L(G)$ of the graph $G = (V, E)$. Fiedler (15) observed that the second eigenvector of L represents a good measure of the connectivity of the graph G. This led to a recursive implementation referred to as recursive spectral bisection (RSB) (16). Other spectral heuristics combining several eigenvectors of L with quadrisection and octasection implementations are proposed and discussed in Ref. 17. The performance of spectral heuristics is presented in Refs. 10 and 18.

Coordinate Axis Splitting. This is another class of enumerative schemes whose main characteristic is that they ignore the connectivity information of the mesh graph G. They are based on coordinate sorting and partitioning along cartesian, polar, and symmetric inertial axis of the graph G. A comprehensive evaluation of these heuristics is reported in Ref. 10, while Refs. 19 and 20 review the underlying ideas of these strategies.

Deterministic Optimization Heuristics. The mesh partitioning problem can be formulated as a constrained or unconstrained optimization problem. This set of heuristics is applied to solve these associated optimization problems. The basis of most of them is the so-called Kernighan and Lin (K–L) algorithm. A detailed review of these class of strategies together with the description of an efficient improvement of the K–L algorithm for mesh/grid can be found in Ref. 21.

Stochastic Optimization Heuristics. Another class of heuristics is based on stochastic techniques such as simulated annealing and Hopfield neural networks. Their application and modification for the partitioning of finite element mesh graph has been studied by several authors (20,22). Although these techniques usually generate more accurate solutions to the mesh partitioning problem, tend to be computationally very intensive.

Hybrid Multilevel Heuristics. Many of the above-described schemes can be accelerated using a multilevel technique. The main idea, whose popularity increased lately, is to replace the graph by a coarser graph with many fewer nodes, partition the coarser graph, and use the coarse partition as a starting guess to obtain a partition of the original graph. The coarse graph may in turn be partitioned by means of the same algorithm recursively; a yet coarser graph is partitioned to get a starting guess, and so on. Such recursiveness allows the use of different heuristic schemes on a different level, resulting in significantly more efficient implementations. This approach, whose typical representative is the multilevel version of RSB (MRSB) (18), been extensively used in modern graph partitioning software tools which we review next. Several tools have been developed to incorporate the above algorithmic infrastructure. Next we briefly comment on five of the most recent and well-known ones. For a comprehensive review of these software tools see Ref. 23.

The //ELLPACK system has a graphical tool (DOMAIN DECOMPOSER) that allow users to obtain and display automatic decompositions by a variety of heuristics for two- and three-dimensional meshes/grids. The user can either modify interactively these decompositions or specify his own. The current version supports both element- and nodewise partitionings using most of the heuristics described above. This tool is completely integrated with the //ELLPACK problem-solving environment, and thus it supports all the parallel discretization and solution modules currently available in the //ELLPACK library.

The CHACO graph partitioning software consists of a library that realizes a variety of partitioning algorithms including spectral bisection, quadrisection, and octasection, the inertial method, variations of the K–L algorithm, and multilevel partitions. In addition, it intelligently embeds the partitions it generates into several different interconnection topologies. It also provides easy access to Fiedler's eigenvectors and advanced schemes for improving data locality. CHACO has been interface, through a GUI to //ELLPACK.

Another system that has been integrated into //ELLPACK is the METIS unstructured graph partitioning tool which implements various multilevel algorithms. There are three basic steps for these algorithms: (1) Collapse vertices of original graph G to coarsen it down to a few hundred vertices, (2) compute a minimum edge-cut bisection of the coarsen graph which is assumed to contain information for intelligently enforcing a balanced partition, and (3) project back to the original graph by periodically further improving partitions using a local refinement heuristic.

A technique similar to the above methodology follows the PARTY partitioning library. It provides a variety of methods for global and local partitioning and offers the option of either (1) partitioning the graph into two parts and then applying a recursive procedure or (2) directly partitioning the graph in the required number of parts. PARTY has been also incorporated into //ELLPACK. Another modern partitioning software package worth mentioning is JOSTLE which can be also used to repartition existing partitions, such as those driving from adaptive refined meshes.

Parallel PDE Software Packages

We conclude this section by briefly presenting in Table 2 some of the software systems publicly available for solving PDE problems on modern (mostly distributed memory) parallel computers. In its first column we list the acronyms, the principal investigators, and their affiliation. The second column describes the class of PDE problems targeted, and the third column gives the parallelization methodologies and the software libraries for communication used in the implementation. The last column deals with the software languages and the floating point arithmetic associated. More information can be obtained from their web servers whose URL addresses are also listed.

PARALLEL DENSE LINEAR ALGEBRAIC SOLVERS

One of the most typical representatives of dense linear algebra solvers is the LU decomposition algorithm. In this section we first review some of the techniques used to parallelize the LU and LU-like factorizations for dense linear systems. Next we present, in a unified way, the parallelization methodologies for a class of fast numerical algebraic solvers specifically designed for special types of elliptic PDEs.

Factorization Methods

The goal of the LU decomposition is to factor an $n \times n$ matrix A into a lower triangular matrix L and an upper triangular matrix U. This factorization is certainly one of the most used of all numerical linear computations. The classical LU factorization can be expressed in terms of any of the three levels of the BLAS (24,25), and techniques needed to achieve high performance for both shared and distributed memory systems have been considered in great detail in the literature.

We consider first some of the approaches used in the literature for implementing the LU factorization of a matrix $A \in \Re^{n\times n}$ on shared memory multiprocessors in which each processor is either of vector or RISC architecture. To simplify the discussion of the effects of hierarchical memory organization, we move directly to the block versions of the algorithms. Throughout the discussion, ω denotes the blocksize used and the more familiar BLAS2 (24)-based versions of the algorithms can be derived by setting $\omega = 1$. Four different organizations of the computation of the classical LU factorization without pivoting are presented with emphasis on identifying the computational primitives involved in each. The addition of partial pivoting is then considered and a block generalization of the LU factorization (L and U being block triangular) is presented for use with diagonally dominant matrices.

There are several ways to organize the computations for calculating the LU factorization of a matrix. These reorgani-

Table 2. List of Software Packages for Parallel PDE Computations

Package	Applicability	Parallelism	Software
COGITO *Smedsaas, Thun, Wahlund,* Uppsala University	Time-dependent PDES	**Commun. libs:** NX, MPI, PVM, MPL **Methodology:** on-line	**Callable:** f90 **Arithmetic:** Real
URL: http://www.tdb.uu.se/research/swtools/cogito.html			
DIFFPACK *Bruaset, Cai, Langtangen, Tveiko,* University of Oslo	General 1–3 dim DEs	**Commun. libs:** custom **Methodology:** on-line	**Callable:** GUI, C++ **Arithmetic:** Real
URL: http://www.nobjects.com/prodserv/diffpack/			
//ELLPACK *Houstis, Rice et al.,* Purdue University	General 1–3 dim DEs	**Commun. libs:** NX, MPI, PVM, NX ... **Methodology:** on-line, off-line, new	**Callable:** GUI, WEB, f77 **Arithmetic:** Real
URL: http://www.cs.purdue.edu/ellpack/ellpack.html			
PETSc *Balay, Gropp, McInnes, Smith,* Argonne National Lab	General 1–3 dim DEs	**Commun. libs:** MPI **Methodology:** on-line, off-line	**Callable:** f77, C/C++ **Arithmetic:** Real, Complex
URL: http://www.mcs.anl.gov/petsc/petsc.html			
PINEAPL *Derakhshan, Hammarling,* NAG	2–3 dim Poisson, Helmholtz	**Commun. libs:** PVM, MPI **Methodology:** on-line	**Callable:** f77 **Arithmetic:** Real, Complex
URL: http://extweb.nag.co.uk/projects/PINEAPL.html			
SUMAA3D *Freitag, Gooch, Jones, Plassmann,* Argonne National Lab	General 2–3 dim DEs	**Commun. libs:** MPI **Methodology:** on-line	**Callable:** f77 **Arithmetic:** Real, Complex
URL: http://www.mcs.anl.gov/Projects/SUMAA/			
TUCHEM *Rame, Soucie, Klie, Wheeler,* TICAM,	Application specific PDEs	**Commun. libs:** MPI **Methodology:** on-line, new	**Callable:** f77 **Arithmetic:** Real, Complex
URL: http://www.ticam.utexas.edu/Groups/SubSurfMod/software.html			
VECFEM *Grosz, Schoenauer, Weis,* University of Karlsruhe	General 2–3 dim boundary value problems	**Commun. libs:** MPI, NX **Methodology:** on-line	**Callable:** GUI, f77 **Arithmetic:** Real, complex
URL: http://www.uni-karlsruhe.de/ vecfem/vecfem2.html			

zations are typically listed in terms of the ordering of the nested loops that define the standard computation. The essential differences between the various forms are: the set of computational primitives required, the distribution of work among the primitives, and the size and shape of the subproblems upon which the primitives operate. Since architectural characteristics can favor one primitive over another, the choice of computational organization can be crucial in achieving high performance. Of course, this choice in turn depends on a careful analysis of the architecture/primitive mapping.

Version 1 of the algorithm assumes that at step i the LU factorization of the leading principal submatrix of dimension $(i-1)\omega$, $A_{i-1} = L_{i-1}U_{i-1}$, is available. The next ω rows of L and ω columns of U are computed during step i to produce the factorization of the leading principal submatrix of order $i\omega$. Clearly, after $k = n/\omega$ such steps the factorization $LU = A$ results. The basic step of the algorithm can be deduced by considering the following partitioning of the factorization of the matrix $A_i \in \Re^{i\omega \times i\omega}$:

$$A_i = \begin{pmatrix} A_{i-1} & C \\ B^T & H \end{pmatrix} = \begin{pmatrix} L_{i-1} & 0 \\ M^T & L_2 \end{pmatrix} \begin{pmatrix} U_{i-1} & G \\ 0 & U_2 \end{pmatrix}$$

where H is a square matrix of order ω and the rest of the blocks are dimensioned conformally. The basic step of the algorithm consists of four phases depicted in Table 3. Clearly, repeating this step on successively larger submatrices will produce the factorization of $A \in \Re^{n\times n}$.

Version 2 of the algorithm assumes that the first $\xi = (i-1)\omega$ columns of L and ξ rows of U are known at the start of step i, and it also assumes that the transformations necessary to compute this information have been applied to the submatrix $A^i \in \Re^{n-\xi\times n-\xi}$ in the lower right-hand corner of A that has yet to be reduced. The algorithm proceeds by producing the next ω columns and rows of L and U, respectively, and computing A^{i+1}. This is a straightforward block generalization of the standard rank-1-based Gaussian elimination algorithm. Assume that the factorization of the matrix $A^i \in \Re^{n-\xi\times n-\xi}$ is partitioned as follows:

$$A^i = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & A^{i+1} \end{pmatrix}$$

where A_{11} is square and of order ω and the other submatrices are dimensioned conformally. L_{11}, L_{21}, and U_{12} are the desired ω columns and rows of L and U and identity defines A^{i+1}. The basic step of the algorithm is presented in Table 3. Clearly, the updated A_{22} is A^{i+1} and the algorithm proceeds by repeating the four phases involved.

Version 3 of the algorithm can be viewed as a hybrid of the first two versions. Like Version 2, it is assumed that the first

Table 3. The Main Step of Four LU Versions

	Version 1	Version 2
(i)	Solve for G: $C \leftarrow L_{i-1}G = C$	Factor: $A_{11} \leftarrow L_{11}U_{11} = A_{11}$
(ii)	Solve for M: $B \leftarrow U_{i-1}^{\mathrm{T}}M = B$	Solve for L_{21}: $A_{21} \leftarrow U_{11}^{\mathrm{T}}L_{21}^{\mathrm{T}} = A_{21}^{\mathrm{T}}$
(iii)	$H \leftarrow H - M^{\mathrm{T}}G$	Solve for U_{12}: $A_{12} \leftarrow L_{11}\mathrm{U}_{12} = A_{12}$
(iv)	Factor $H \leftarrow L_2U_2 = H$	$A_{22} \leftarrow A_{22} - L_{21}U_{12}$
	Version 4	**Version 5**
(i)	Solve for M: $\tilde{A}_1 \leftarrow L_{11}M = \tilde{A}_1$	$A_{11} \leftarrow A_{11}^{-1}$
(ii)	$[\tilde{A}_2^T, \tilde{A}_3^T]^{\mathrm{T}} \leftarrow [\tilde{A}_2^T, \tilde{A}_3^T]^{\mathrm{T}} - L_{21}M$	$A_{21} \leftarrow L_{21} = A_{21}A_{11}$
(iii)	Factor: $\tilde{A}_2 \leftarrow \tilde{L}\tilde{U} = \tilde{A}_2$	$A_{22} \leftarrow B = A_{22} - L_{21}A_{12}$
(iv)	Solve for G: $\tilde{A}_3 \leftarrow \tilde{U}^T G^{\mathrm{T}} = \tilde{A}_3^{\mathrm{T}}$	Proceed recursively on matrix B

The arrow is used to represent the portion of the array which is overwritten by the new information obtained in each phase.

$(i - 1)\omega$ columns of L and rows of U are known at the start of step i. It also assumes, like Version 1, that the transformations that produced these known columns and rows must be applied elements of A which are to be transformed into the next ω columns and rows of L and U. As a result, Version 3 does not update the remainder of the matrix at every step. Consider the factorization

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$

where A_{11} is a square matrix of order $(i - 1)\omega$ and the rest are partitioned conformally. By our assumptions, L_{11}, L_{21}, U_{11}, and U_{12} are known and the first ω columns of L_{22} and the first ω rows of U_{22} are to be computed. Since Version 3 assumes that none of the update $A_{22} \leftarrow A_{22} - L_{21}U_{12}$ has occurred in the first $i - 1$ steps of the algorithm, the first part of step i is to perform the update to the portion upon which the desired columns of L_{22} and rows of U_{22} depend. This is then followed by the calculation of the columns and rows. To derive the form of the computations, suppose that the update of A_{22} and its subsequent factorization are partitioned as

$$A_{22} \leftarrow \begin{pmatrix} H & C^{\mathrm{T}} \\ B & \tilde{A}_{22} \end{pmatrix} = \begin{pmatrix} \hat{H} & \hat{C}^{\mathrm{T}} \\ \hat{B} & \hat{A}_{22} \end{pmatrix} - L_{21}U_{12}$$

with

$$\begin{pmatrix} H & C^{\mathrm{T}} \\ B & \tilde{A}_{22} \end{pmatrix} = \begin{pmatrix} \tilde{L}_{11} & 0 \\ \tilde{L}_{21} & \tilde{L}_{22} \end{pmatrix} \begin{pmatrix} \tilde{U}_{11} & \tilde{U}_{12} \\ 0 & \tilde{U}_{22} \end{pmatrix}$$

where H and $\hat{H}$ are square matrices of order ω and the other submatrices are dimensioned conformally. Step i then has two major phases: Calculate H, B, and C; and calculate $\tilde{L}_{11}$, $\tilde{L}_{21}$, $\tilde{U}_{11}$, and $\tilde{U}_{12}$. As a result, at the end of stage i, the first iω rows and columns of the triangular factors of A are known. Let $L_{21} = [M_1^{\mathrm{T}}, M_2^{\mathrm{T}}]^{\mathrm{T}}$ and $U_{12} = [M_3, M_4]$, where M_1 and M_3 consist of the first ω rows and columns of the respective matrices. The first phase of step i computes

1. $[H^{\mathrm{T}}, B^{\mathrm{T}}]^{\mathrm{T}} \leftarrow [H^{\mathrm{T}}, B^{\mathrm{T}}]^{\mathrm{T}} = [\hat{H}^{\mathrm{T}}, \hat{B}^{\mathrm{T}}]^{\mathrm{T}} - L_{21}M_3$
2. $C \leftarrow C^{\mathrm{T}} = \hat{C}^{\mathrm{T}} - M_1M_4$

In the second phase, the first ω rows and columns of the factorization of the updated A_{22} are then given by the following:

1. Factor: $H \leftarrow \tilde{L}_{11}\tilde{U}_{11} = H$
2. Solve for $\tilde{L}_{21}$: $B \leftarrow \tilde{U}_{11}^{\mathrm{T}}\tilde{L}_{21}^{\mathrm{T}} = B^{\mathrm{T}}$
3. Solve for $\tilde{U}_{12}$: $C \leftarrow \tilde{L}_{11}\tilde{U}_{12} = C^{\mathrm{T}}$

Version 4 of the algorithm assumes that at the beginning of step i the first $(i - 1)\omega$ columns of L and U are known. Step i computes the next ω columns of the two triangular factors. Consider the factorization

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$

where A_{11} is a square matrix of order $(i - 1)\omega$ and the rest are partitioned conformally. By our assumptions, L_{11}, L_{21}, and U_{11} are known.

Let L_ω, U_ω, and A_ω be the matrices of dimension $n \times \omega$ formed of the first ω columns of $[0, L_{22}^{\mathrm{T}}]^{\mathrm{T}}$, $[U_{12}^{\mathrm{T}}, U_{22}^{\mathrm{T}}]^{\mathrm{T}}$, and $[A_{12}^{\mathrm{T}}, A_{22}^{\mathrm{T}}]^{\mathrm{T}}$, respectively. [These are also columns $(i - 1)\omega + 1$ through $i\omega$ of L, U, and A.] Consider the partitioning

$$L_\omega = \begin{pmatrix} 0 \\ \tilde{L} \\ G \end{pmatrix}, \qquad U_\omega = \begin{pmatrix} M \\ \tilde{U} \\ 0 \end{pmatrix}, \qquad A_\omega = \begin{pmatrix} \tilde{A}_1 \\ \tilde{A}_2 \\ \tilde{A}_3 \end{pmatrix}$$

where $\tilde{L}$, $\tilde{U}$, and $\tilde{A}_2$ are square matrices of order ω with $\tilde{L}$ and $\tilde{U}$ lower and upper triangular, respectively. Step i calculates L_ω and U_ω by applying all of the transformations from steps 1 to $i - 1$ to A_ω and then factoring a rectangular matrix. Specifically, step i comprises the computations depicted in Table 3.

Partial pivoting can be easily added to Versions 2, 3, and 4 of the algorithm. Step i of each of the versions requires the LU factorization of a rectangular matrix $M \in \Re^{h \times \omega}$, where $h = n - (i - 1)\omega$. Specifically, step i computes

$$M = \begin{pmatrix} M_1 \\ M_2 \end{pmatrix} = \begin{pmatrix} \hat{L}_{11} \\ \hat{L}_{21} \end{pmatrix} \hat{U}_{11}$$

where $\hat{L}_{11}$ and $\hat{U}_{11}$ are, respectively, lower and upper triangular matrices of order ω. In the versions above without pivoting, this calculation could be split into two pieces: the factorization of a system of order ω, $\hat{L}_{11}\hat{U}_{11} = M_1$; and the solution of a triangular system of order ω with $h - \omega$ right-hand sides. When partial pivoting is added, these computations at each step cannot be separated and are replaced by a single primitive which produces the factorization of a rectangular matrix with permuted rows, that is,

$$PM = P\begin{pmatrix} M_1 \\ M_2 \end{pmatrix} = \begin{pmatrix} \hat{L}_{11} \\ \hat{L}_{21} \end{pmatrix} \hat{U}_{11}$$

where P is a permutation matrix. This primitive is usually cast as a BLAS2 version of one of the versions above. Note, however, a fundamental difference compared to the nonpivoting versions. The ability to split the factorization of the tall matrix into smaller BLAS3 (25)-based components in the latter case has benefits with respect to hierarchical memory usage, since ω is usually taken so that such systems fit in cache or local memory. In the case of pivoting, these operations are performed via BLAS2 primitives repeatedly updating a matrix which cannot be kept locally. As a result, the arithmetic component of time and the data transfer overhead both increase. In fact, a conflict between their reductions occurs. This situation is similar to that in Version 5 presented below along with a solution.

The information contained in the permutations associated with each step, P_i, can be applied in various ways. For example, the permutation can be applied immediately to the transformations of the previous steps, which are stored in the elements of the array A to the left of the active area for step i, and to the elements of the array A which have yet to reach their final form, which, of course, appear to the right of the active area for step i. The application to either portion of the matrix may also be delayed. The update of the elements of the array which have yet to reach their final form could be delayed by maintaining a global permutation matrix which is then applied to only the elements required for the next step. Similarly, the application to the transformations from steps 1 through $i - 1$ could be suppressed and the P_i could be kept separately and applied incrementally in a modified forward and backward substitution routine.

In some cases it is possible to use a block generalization (Version 5) of the classical LU factorization in which L and U are lower and upper block triangular matrices, respectively. The use of such a block generalization is most appropriate when considering systems which do not require pivoting for stability—for example, diagonally dominant or symmetric positive definite. This algorithm decomposes A into a lower block triangular matrix L_ω and an upper block triangular matrix U_ω with blocks of the size ω by ω (it is assumed for simplicity that $n = k\omega$, $k > 1$). Assume that A is diagonally dominant and consider the factorization

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ 0 & B \end{pmatrix}$$

where A_{11} is a square matrix of order ω. The block LU algorithm is given in Table 3, where statements (i) and (ii) can be implemented in several ways (26).

If the Gauss–Jordan kernel is used, as is assumed below, the block LU algorithm is more expensive by a factor of approximately $(1 + 2/k^2)$ than the classical LU factorization, which requires about $2n^3/3$ operations. In this form, the above block algorithm uses three primitives: a Gauss–Jordan inversion (or LU decomposition), $A \leftarrow AB$, and a rank-ω update.

Note that when $\omega = 1$ this form of the algorithm becomes the BLAS2 version based on rank 1 updates. As with Versions 1–4, which produce the classical LU factorization, the computations of Version 5 can be reorganized so that different combinations of BLAS3 primitives and different *shapes* of submatrices are used (26).

For distributed-memory implementations we consider the two basic storage schemes: storage of A by rows and by columns. These row storage cases lead to the so-called *Row Storage with Row Pivoting* (RSRP) algorithm and the *Column Storage with Row Pivoting* (CSRP) scheme. Gaussian elimination with pairwise pivoting is an alternative to LU factorization, which is attractive on a variety of distributed memory architectures since it introduces parallelism into the pivoting strategy. Pairwise pivoting can also be useful on shared memory machines to break the bottleneck caused by partial pivoting discussed earlier.

Rapid Direct Solvers

Often the physical problem and its mathematical model possess properties that can be exploited to design fast numerical solution procedures. Such methods exist for special types of elliptic PDEs and are collectively known as *rapid elliptic solvers* (RES), though the more specific term *fast Poisson solvers* is also used. These methods consist primarily of noniterative techniques that achieve significantly lower complexities than traditional solution methods. Several RES have their roots in classical analytical techniques such as separation of variables and the use of Fourier expansions (27,28). For a detailed examination of the topic of this article we also refer the reader to the Ref. 29.

Here we concentrate on parallel aspects of RES algorithms. The model problem that will be used is Poisson's equation with Dirichlet boundary conditions on the unit square:

$$-(V_{xx} + V_{yy}) = F \quad \text{for } (x, y) \in \Omega \equiv [0, 1] \times [0, 1] \text{ and } V(x, y) \text{ given on } \partial\Omega \qquad (1)$$

It is easy, in theory, to solve Eq. (1). For example, when the boundary conditions are homogeneous, the solution can be written as an integral of the product of the right-hand side and Green's function for the Laplacian on the unit square (27). However, the evaluation of this integral is not practical in principle and we have to resort to alternative numerical techniques.

We use tensor (Kronecker) products to describe the method. These capture the parallel aspects of the method and highlight the manner in which multidimensional data arrays are manipulated by applying operations on lower-dimensional data slices (30,31).

The discretization of Eq. (1) with second-order differences on a uniformly spaced rectangular grid and the incorporation of the boundary conditions leads, under natural ordering (32), to the order $N = n_x n_y$ linear system:

$$-(I_{n_y} \otimes T_x + T_y \otimes I_{n_x})u = f \qquad (2)$$

where T_x and T_y are symmetric tridiagonal matrices of orders n_x and n_y, respectively, that correspond to the finite-difference approximation of second-order derivative operators. We next describe some important characteristics of T_x (which is diagonalizable) with the understanding that the corresponding results hold for T_y. For the model problem, $T_x := \text{trid}_{n_x}[-1, 2, -1]$ and $T_y := \text{trid}_{n_y}[-\alpha, 2\alpha, -\alpha]$, $\alpha := h_y^2/h_x^2$, where $h_x = (n_x + 1)^{-1}$ and $h_y = (n_y + 1)^{-1}$ are the discretization steps in the x and y directions, respectively.

When $n := n_x = n_y$ the best sequential complexity for solving the model problem via direct methods is $O(n^2 \log n)$. (Throughout this section, log denotes $\log_2$.) The RES that we describe solve Eq. (2) with asymptotic operation counts $O(n^2 \log^k n)$ and parallel complexities $T(\log^k n, n^m)$, where $T(\tau, P)$ denotes the complexity of an algorithm that requires time $O(\tau)$ when P processors are used and where k and m are small constants. RES can achieve parallel time as low as $O(\log n)$ on certain computational models of $O(n^2)$ processors. It has been seen that some of the methods we describe are asymptotically time optimal. Nevertheless, a useful evaluation of performance requires more realistic computational models. Unfortunately, a unifying model for the wide spectrum of parallel architectures has yet to be established.

One unifying aspect of all the methods described in this article is that they are composed of two basic computational primitives: (1) the solution of banded (mostly tridiagonal) systems of equations and (2) the discrete Fourier transform. It is thus possible to provide a first approximation to the performance of these methods from performance information about the primitives. Other advantages of this formulation include the easier and more flexible, and hence faster, development of codes for a range of architectures, along with better identification of weaknesses in current architectures and their supporting software (26). Software design also becomes easier using, for instance, primitives that hide most of the architectural details.

The RES described below have a multiphase structure; each phase consists of the application of one or more instances of a computational primitive. Synchronization is enforced between phases. During a single phase, in addition to the parallelism available within the algorithm implementing the underlying primitive, another major source of parallelism can be found in the independent application of multiple instances of the computational primitive—for example, the application of the same matrix operation on several vectors. As in the case of the BLAS (26), we expect that the best RES will combine these two sources of parallelism.

Matrix decomposition (MD) is one of the most important RES methods (33,34). If Q_x denotes the matrix with elements $[Q_x]_{jk} = \sin(\pi jk/(n_x + 1))$ $(1 \le k, j \le n_x)$, then $Q_x^{-1}T_xQ_x = \Lambda_x$ denotes the diagonal matrix of the eigenvalues $\lambda_j = 2 - 2\cos(\pi j/(n_x + 1))$ $(1 \le j \le n_x)$ of T_x. The inverse is readily available and satisfies $Q_x^{-1} = (2/n_x)Q_x$. Since Q_x is the matrix representation of the discrete sine transform operator, its application on a vector of length n_x can be accomplished using an FFT algorithm, at a cost of $O(n_x \log n_x)$ operations instead of $O(n_x^2)$. This remark is at the root of the low-complexity solvers described here.

Premultiplying Eq. (2) by $I_{n_y} \otimes Q_x^{-1}$, we obtain

$$-I_{n_y} \otimes Q_x^{-1}(I_{n_y} \otimes T_x + T_y \otimes I_{n_z})(I_{n_y} \otimes Q_x)(I_{n_y} \otimes Q_x^{-1})u = I_{n_y} \otimes Q_x^{-1} f$$

hence,

$$-(I_{n_y} \otimes \Lambda_x + T_y \otimes I_{n_x})(I_{n_y} \otimes Q_x^{-1})u = I_{n_y} \otimes Q_x^{-1} f \qquad (3)$$

Matrix $I_{n_y} \otimes \Lambda_x + T_y \otimes I_{n_x}$ in Eq. (3) is block tridiagonal with diagonal nonzero blocks and can be reordered to block diagonal form with tridiagonal nonzero blocks. This rearrangement is achieved through an order $N = n_xn_y$ permutation matrix Π_{N,n_x} whose action on an order N vector x is defined as follows:

$$y := \Pi_{N,n_x} x = [x(1 : n_x : N), x(2 : n_x : N), \ldots, x(n_x : n_x : N)]^{\mathrm{T}}$$

If x is organized as a two-dimensional array with n_x rows and n_y columns, then y is obtained after transposing x and numbering its elements in column major order. Two important properties are that $\Pi_{N,n_x}\Pi_{N,n_y} = I_{n_xn_y}$ and $\Pi_{N,n_x}(C \otimes I_{n_x})\Pi_{N,n_y} = I_{n_x} \otimes C$ for any order n_y matrix C. Therefore Eq. (3) can be rewritten as

$$-\Pi_{N,n_x}(I_{n_y} \otimes \Lambda_x + T_y \otimes I_{n_x})(I_{n_y} \otimes Q_x^{-1})u = \Pi_{N,n_x}(I_{n_y} \otimes Q_x^{-1}) f \qquad (4)$$

from which follows that the solution can be expressed as

$$u = -(I_{n_y} \otimes Q_x)\Pi_{N,n_y}(\Lambda_x \otimes I_{n_y} + I_{n_z} \otimes T_y)^{-1}\Pi_{N,n_x}(I_{n_y} \otimes Q_x^{-1}) f \qquad (5)$$

In the sequel we will assume that whenever the operator $(I_r \otimes C_s)$ is applied on an rs vector f, the vector f is partitioned into r subvectors of order s each. Algorithm MD solves Eq. (2) using the formulation in Eq. (5):

Algorithm MD (Matrix Decomposition)

```
1. Solve: $Q_x y_j = f_j$ $(1 \le j \le n_y)$.
2. Permute: $\bar{y} = \Pi_{N,n_x} y$.
3. Solve: $(T_y + \lambda_i^{(x)} I)\hat{y}_i = \bar{y}_i$ $(1 \le i \le n_x)$.
4. Permute: $y = \Pi_{N,n_y} \hat{y}$.
5. Compute $u_j = Q_x y_j$ $(1 \le j \le n_y)$.
```

This algorithm consists of three major computational steps: (i) the computation of n_y Fourier transforms of length n_x each, (ii) the solution of n_x tridiagonal systems of order n_y each, and (iii) the computation of n_y inverse Fourier transforms of length n_x each. Regarding the cost of computing the entries of Q_x, whenever this is necessary, we take that it is amortized over several calls to MD, so we do not consider it in the cost estimates. Let us represent steps i–iii by the letter sequence *FTF*, where F and T represent the application of several independent computational primitives for the Fourier transform and the solution of tridiagonal systems, respectively.

Since each step of MD calls for multiple instances of the tridiagonal solver and the FFT primitives, there are two basic parallel approaches. One is to use a serial algorithm to compute each instance of a computational primitive; we use the superscript s to denote such an implementation. This approach requires that prior to the call to a primitive, all the necessary data are available in the memory of each executing processor. Since no communication is required across the processors during each of the steps, this method offers large grain parallelism. In the second approach, a parallel algorithm is used to evaluate each primitive. We use the superscript p to denote such implementations. Each processor will have to deal with segments from one or more independent data sets. Communication is thus necessary across the processors during execution; hence this method offers medium and fine grain parallelism. In the literature, the above two methods have been called the "distributed data" and the "distributed algorithm" approaches.

We define four practical instances of the MD algorithm for the two-dimensional model problem: $F^sT^sF^s$, $F^sT^pF^s$, $F^pT^pF^p$, and $F^pT^sF^p$, which we review next.

$F^sT^sF^s$. In the first step of this method, each processor calls an algorithm to apply length n_x fast Fourier transforms (FFTs) on data from n_y/p grid rows. Subsequently, each processor calls an algorithm to solve n_x/p tridiagonal systems of order n_y each. The right-hand sides of these systems correspond to grid columns of data. Finally, each processor calls an algorithm to apply length n_x inverse FFTs on n_y/p grid rows of data. Processors must synchronize between steps (35). One advantage of this method is that no parallel algorithm is required to implement the computational primitives. We can thus view this as a parallel method synthesized out of serial, off-the-shelf pieces (35). Nevertheless, the implementation has to be done carefully; otherwise performance will degrade due to excessive data traffic between the processors and the memory system (36).

On shared memory architectures no explicit data transposition is necessary. One cause of performance loss is bank conflicts originating from non-unit stride memory references in one of the steps. On distributed memory machines, there is a need for a matrix transposition primitive in order to implement the stride permutation described earlier and bring into each processor's local memory one or more columns of data computed during step F^s. Parallel algorithms and implementations for matrix transposition can be found in Ref. 31, Chap. 3.

$F^sT^pF^s$. The first and last steps of this method are as for $F^sT^sF^s$. Immediately prior to the second step, each processor has immediate access to n_y/p components from each of the n_x tridiagonal systems; hence explicit transposition is unnecessary (37). These are solved using one of the parallel algorithms described previously. The method also exploits any of the system's vector and communication pipelining capabilities.

$F^pT^pF^p$. This method is uses parallel algorithms for each of the computational primitives. Trivially, T^p and F^p require less parallel time than T^1 and F^1, respectively; therefore, this method can achieve the best parallel arithmetic complexity among all other MD methods. The performance of this algorithm on actual systems is significantly affected by the time spent in memory accesses and interprocessor communication (37). To achieve optimal arithmetic complexity, it is critical to use algorithms of commensurate (logarithmic) complexity for each step. It thus becomes necessary to use a tridiagonal solver such as cyclic reduction, instead of parallel substructured Gaussian elimination.

$F^pT^sF^p$. The parallel arithmetic complexity of this method for square grids is $O(n)$. The methods above can be generalized to handle three-dimensional problems; in that case, however, there is a larger number of parallel implementations that can be defined ($F^sF^sT^sF^sF^s$, $F^sF^pT^sF^pF^s$, and $F^sF^sT^pF^sF^s$). The methodology that we used above to present the MD implementations will be applied below to other RES.

The MD method is based on the fact that using fast transforms and row and column permutations, the coefficient matrix can be brought into block diagonal form with tridiagonal blocks in the diagonal. The next method uses the fact that this block diagonal matrix can be further reduced into diagonal form by means of fast transforms. To see this, premultiply both sides of Eq. (2) by $Q_y^{-1} \otimes I_{n_x}$ to obtain

$$-(I_{n_y} \otimes \Lambda_x + \Lambda_y \otimes I_{n_x})(Q_y^{-1} \otimes I_{n_x})(I_{n_y} \otimes Q_x^{-1})u = (Q_y^{-1} \otimes I_{n_x})(I_{n_y} \otimes Q_x^{-1})f \quad (6)$$

Hence the solution can be expressed as

$$u = (I_{n_y} \otimes Q_x)\Pi_{N,n_y}(I_{n_x} \otimes Q_y)\Pi_{N,n_x}(I_{n_y} \otimes \Lambda_x + \Lambda_y \otimes I_{n_x})^{-1} \quad (7)$$

$$\Pi_{N,n_y}(I_{n_x} \otimes Q_y^{-1})\Pi_{N,n_x}(I_{n_y} \otimes Q_x^{-1})f \quad (8)$$

We call the methods based on the formulation in Eq. (7) "full matrix diagonalization" (FMD) methods. The terms "multiple Fourier" or "complete Fourier" methods have also been used.

Algorithm FMD

```
1.  $y_j = Q_x^T f_j$, $(1 \le j \le n_y)$, $\bar{y} = \Pi_{N,n_x} y$.
2a. $\hat{y}_i = Q_y^T \bar{y}_j$, $(1 \le i \le n_x)$, $\bar{y} = \Pi_{N,n_y} \hat{y}$.
2b. $\hat{y}_{j,i} = (\lambda_j^{(y)} + \lambda_i^{(x)})^{-1} \hat{y}_{j,i}$, $(1 \le i \le n_x,\ 1 \le j \le n_y)$, $\bar{y} = \Pi_{N,n_x} \hat{y}$.
3.  $y_i = Q_y \bar{y}_i$, $(1 \le i \le n_x)$, $\hat{y} = \Pi_{N,n_y} y$.
4.  $u_j = Q_x \hat{y}_j$, $(1 \le j \le n_y)$.
```

FMD methods can be represented by the pattern $FF\Delta FF$, where Δ denotes element by element division of two length N vectors and F denotes the application of Fourier transforms on rows and columns of data. Synchronization is needed between phases (see Refs. 31, 38, and 39).

The FMD approach trades the tridiagonal solvers of MD with Fourier transforms. This is not necessarily cost effective; for instance, on a uniprocessor, computing the FFT is more expensive than solving a tridiagonal system. On parallel architectures, however, the situation can change, since both T^p and F^p can be implemented in $O(\log n)$ parallel arithmetic operations. It is easy to see that an implementation of $F^pF^p\Delta F^pF^p$ can achieve $T(\log(n_xn_y), n_xn_y)$ parallel arithmetic complexity. Implementations of the $F^pF^p\Delta F^pF^p$ method can be faster than the other solvers presented in this article (38, Algorithm PARAFT).

The method of block cyclic reduction (BCR) (40–42) forms the basis of a popular package for the rapid solution of elliptic PDEs, called FISHPAK (43). We first present the algorithm and review the idea behind its parallelization. For simplicity let $n_y = 2^k - 1$ and consider the equations for three adjacent block rows (it is assumed that $u_{-1} = u_{2^k+1} = 0$ and that u_0 and u_{2^k} have been absorbed in the right-hand side):

$$\begin{aligned} -u_{2i-2} + Au_{2i-1} - u_{2i} &= f_{2i-1} \\ -u_{i-1} + Au_{2i} - u_{i+1} &= f_{2i} \\ -u_{2i} + Au_{2i+1} - u_{2i+2} &= f_{2i+1} \end{aligned}$$

Multiplying the first and last equations by A and adding them to the second one, we obtain

$$-u_{2i-2} + (A^2 - 2I)u_{2i} - u_{2i+2} = f_{2i-1} + Af_{2i} + f_{2i+1} \quad (9)$$

We call this a reduction step, and by letting $i = 1, \ldots, 2^{k-1} - 1$ we form the block system

$$\begin{bmatrix} A^{(1)} & -I & & \\ -I & A^{(1)} & & \\ & \ddots & & -I \\ & & -I & A^{(1)} \end{bmatrix} \begin{bmatrix} u_2 \\ u_4 \\ \vdots \\ u_{2^k-2} \end{bmatrix} = \begin{bmatrix} f^{(1)}_{2\cdot 1} \\ \vdots \\ \vdots \\ f^{(1)}_{2(2^{k-1}-1)} \end{bmatrix} \quad (10)$$

where $A^{(1)} := A^2 - 2I$ and $f^{(1)}_{2i} = Af_{2i} + f_{2i-1} + f_{2i+1}$.

The system in Eq. (10) is approximately half the size of the original one and only involves even indexed unknown vectors. Once this system has been solved, the remaining unknowns can be obtained from the block diagonal system

$$\begin{bmatrix} A & 0 & & \\ 0 & A & 0 & . \\ & \ddots & \ddots & \ddots \\ & & 0 & A \end{bmatrix} \begin{bmatrix} u_1 \\ u_3 \\ \vdots \\ u_{2^k-1} \end{bmatrix} = \begin{bmatrix} f_1 + u_2 \\ f_3 + u_2 + u_4 \\ \vdots \\ \vdots \\ f_{2^k-1} + u_{2^k-2} \end{bmatrix} \quad (11)$$

To form the right-hand sides of the reduced system [Eq. (10)] we need the products $A[f_2, f_4, \ldots, f_{2^k-2}]$. These can be computed with a sparse by dense matrix multiplication kernel. In turn, the solution of the system in Eq. (11) can be achieved with a kernel that solves a tridiagonal system of equations with multiple right-hand sides $A\tilde{X} = \tilde{B}$, where X, B are of size $n_x \times (2^{k-1})$.

Note that $A^{(1)}$ is a polynomial of degree 2 in A that we denote by $p_2(A) := A^2 - 2I$. To avoid fill-in, it is preferable to express the polynomial in factored form $p_2(A) = (A - 2(\sqrt{2}/2)I)(A + 2(\sqrt{2}/2)I)$. The reduction and factorization process can be repeated until a system consisting of a system for the "middle" unknown vector is left. Unfortunately this process is unstable. The scheme used in practice was proposed by Buneman (40,44). Where the recurrence $A^{(j)} = (A^{(j-1)})^2 - 2I$, $A^{(0)} = A$, it is shown that $A^{(r)} = P_{2^r}(A) = T_{2^r}(A/2)$, where T_{2^r} is the Chebyshev polynomial of the first kind and degree 2^r. Hence all operations involving $A^{(r)}$ are written as operations with polynomials in A. There are several interesting design issues that face the implementor of parallel BCR which are discussed in Refs. 41 and 45–48.

A software package that implements some of the rapid solvers described above is CRAY-FISHPAK (49) (a vectorized and parallelized version of FISHPAK). CRAYFISHPAK contains "driver" and "solver" routines for the Helmholtz equation over regular regions in two or three dimensions in cartesian, cylindrical, or spherical coordinates. One important extension of CRAYFISHPAK is that it contains solvers that are based on FFT-based matrix decomposition as well, in addition to the solvers based on the parallel version of Buneman's BCR. Sweet (49) reported that the FFT-based solvers are preferable since they were 10 to 20 times faster than those of FISHPAK when running on vector machines, while the speedup for parallel BCR was only between 4 and 5.

The FACR (Fourier analysis, cyclic reduction) (38,50–54) family of methods combines BCR and MD. The motivation for the algorithm was to reduce the number of Fourier transforms necessary in the first and fourth steps of the MD algorithm. The key idea is to start with r steps of block reduction to obtain the block tridiagonal system $\text{trid}_{2^{k-r}-1}[-I, A^{(r)}, -I]X^{(r)} = Y^{(r)}$, where $A^{(r)}$ is a matrix polynomial in A. Therefore, if we apply the Fourier transform matrix Q that was used in algorithm MD to diagonalize A, we obtain $Q^TA^{(r)}Q = \Lambda^{(r)}$, where $\Lambda^{(r)}$ is a diagonal matrix with coefficients $P_{2^r}(\lambda_j)$. Hence we apply first the MD algorithm to solve the reduced system and then apply back-substitution steps to recover the remaining unknowns.

PARALLEL SPARSE LINEAR ALGEBRAIC SOLVERS

Very frequently a physical phenomenon or problem involves operators that act locally. This is reflected at the linear algebra level as sparsity in the corresponding linear system. Sparseness therefore is a basic characteristic of many large-scale scientific and engineering computations (Table 4). All three parallel methodologies depicted in Fig. 1 assume the existence of efficient parallel linear sparse solvers implemented on a set of distributed algebraic data structures. The overall efficiency of a linear solver can be enormously increased if the underline algorithm properly exploits the nonzero structure of the associated coefficient matrix. In fact it is still infeasible to solve a large class of important physical problems without exploiting their sparseness. We call a matrix sparse if it is worth to exploit its nonzero structure. In practice, such matrices should have a small constant number of nonzero elements per column/row. In some cases the sparsity can be organized so the dense solvers described in previous sections can be effectively applied. A typical, and important, organized sparseness is when all of the nonzero elements are packed inside a relatively small zone that contains the main diagonal of the matrix. Such matrices are called *banded* and appear frequently in discretizing PDE problems. Unfortunately, sparsity often comes with irregularity and diversity which justifies the existence of the MatrixMarket web server, with nearly 500 sparse matrices from a variety of applications, as well as matrix generation tools and services and the SPARSEKIT software package for manipulating and working with different forms of sparse matrices. Compared to dense computations, sparse matrix computations use more sophisticated data structures and involve more irregular memory reference patterns and therefore are significantly more difficult to effectively parallelize them. Several parallelization approaches using purely deterministic mathematical approaches, heuristics, and run-time and compiler technologies have already been applied with notable success.

Many parallel sparse solvers have been proposed and studied in the literature. Their detailed exposition is beyond the scope of this article. Instead, we discuss and reference those that are already available in the form of software and have been tested for the parallel solution of PDE equations. It is worth noting, however, that there is still no general-purpose sparse software system that exhibits significant efficiency on modern parallel computers. We split the rest of this section into four parts. The first deals with direct sparse solvers, the second

Table 4. List of Software Packages for Parallel Sparse Computations

		General
SPARSEKIT	1994	http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html
MatrixMarket	1996	http://math.nist.gov/MatrixMarket/
		Direct
CAPSS	1996	http://www.netlib.org/scalapack/capss.tar.gz
PSPASES	1997	http://www-users.cs.umn.edu/ mjoshi/pspases/
SPOOLES	1997	http://www.netlib.org/linalg/spooles.1.0.tar.gz
SuperLU	1998	http://www.netlib.org/scalapack/prototype/index.html
		Iterative
AZTEC	1998	http://www.cs.sandia.gov/CRF/aztec1.html
BlockSolve95	1995	http://www.mcs.anl.gov/sumaa3d/BlockSolve/index.html
‖ ITPACK	1996	http://pellpack.cs.purdue.edu/
PCG	1995	http://www.cfdlab.ae.utexas.edu/pcg/index.html
PIM	1997	http://www.mat.ufrgs.br/pim-e.html
P-SPARSLIB	1998	http://www.cs.umn.edu/Research/arpa/p_sparslib/psp-abs.html
		Preconditioning
SPAI	1997	http://lovelace.nas.nasa.gov/NAS/SPAI/download.html
ParPre	1997	http://www.math.ucla.edu/eijkhout/parpre.html

In the first column we have the name of the package, in the second column we have the year of the last release, and in the last column we have the address of the associated web page.

with iterative methods and the third with preconditioning. The last part contains a table which provides links to the web pages of all parallel sparse software packages considered.

Sparse Direct Methods

In the absence of any particular nonzero structure a sparse system needs to be solved by factorizing its coefficient matrix using a Gauss elimination-type algorithm. Such algorithms, unfortunately, generate at run time intermediate results and computational tasks of varying granularity in a rather unpredictable way. Specifically the *LU* factorization algorithm for a sparse matrix A leads to the generation of nonzero elements in the factors L and U at places where A is zero. This phenomenon (known as "fill-in") strongly depends on the ordering of the equations/unknowns of the linear system and might totally destroy the sparseness of A producing almost dense factors. Besides computational and memory overheads, fill-in makes the prediction of the data structures required for the factors L, U impossible without actually performing the factorization beforehead. Due to the above reasons, achieving good efficiency for sparse matrix computation on modern computers is still a challenge. Several excellent review articles already exist on this subject, with Refs. 55 and 56 being the most recent ones.

To simplify the beginning of our discussion we assume that A is well-conditioned (e.g., positive definite) and therefore there is no need for pivoting during the elimination process to avoid round-off error affects. This is a basic assumption which, along with symmetricity on A, is very frequently imposed in sparse factorization studies. Therefore we consider the Cholesky factorization process of $A = LL^T$ which requires the following steps:

1. Order A to achieve sparseness in L.
2. Determine data structures for L by symbolically factorizing the ordered matrix.
3. Compute L by numerically factorizing the ordered matrix.
4. Solve the resulting two triangular systems to compute the solution.

The task of determining an optimum ordering of A that minimizes fill-ins proved to be an NP-complete problem. Therefore various heuristic approaches are currently employed for step 1 above. They use a graph representation of the nonzero structure of the matrices and are based on most of the graph partitioning schemes discussed earlier in the section entitled "Hybrid Multilevel Heuristics." In particular, for almost a decade the *minimum degree* (MD) ordering algorithm was (and to some extent remains so) the only efficient scheme that produces reasonably good orderings. In contrast to the minimum degree that uses a local view of sparseness, the relatively recent *nested dissection* (ND) ordering method is based on a global view and usually produces better orderings for a large class of matrices. Its drawback is that it is difficult to apply it to general matrices effectively. Step 1 usually costs less than the subsequent steps. Nevertheless, one needs to parallelize it if good over all scalabability is desired. For this in the past few years several effective parallel (and serial) extensions/implementations of the MD and ND (both are not easily parallelizable) have been proposed (55). Step 3 consumes most of the computing time. Its main source of parallelism is in performing a set of row or column modifications by using multiple of prior rows columns. A key technique that leads to significant improvement in the performance of parallel sparse direct methods is to recognize that as the elimination progresses, the reduced matrix in sparse Gaussian elimination becomes denser and at some point it is more efficient to switch to dense code. There are two main approaches here that allow the utilization of higher-level BLAS through the entire factorization. These are the frontal and supernodal approaches (56).

CAPPS is a package that implements the above factorization scheme in parallel by assuming that the coefficient matrix is symmetric and positive definite. It further assumes that each unknown is associated with a set of coordinates in Euclidean space (e.g., associated with a mesh point whose coordinates are known). It uses a Cartesian nested dissection ordering to minimize the fill-in and increase parallelism and consists of two disjoint codes, Map and Solve. Map first reads the matrix in a generalized symmetric sparse form and then runs sequential to map columns and nonzero elements to processors by generating a set of input files. Each one of them contains the local to each processor data. Solve is the fully parallel code which runs hostless on each processor, reads the associated input file, and solves the linear system by exploiting the block nonzero structure of the ordered matrix. Specifically the nested dissection ordering creates zero blocks that are preserved during the solution process and that allow the factorization of certain of the nonzero blocks to be done in parallel. PSPASES is another parallel sparse solver restricted to symmetric and positive definite linear systems. In contrast to CAPPS, it is fully parallel utilizing ParaMETRIS to determine a fill-in reducing ordering. Both PSPASES and CAPSS are MPI-based implementations written in Fortran/C.

Many important applications do not lead to either symmetric or positive definite matrices. Therefore one needs to switch from Cholesky to *LU* factorization Partial pivoting to achieve numerical stability in the last two steps of the above given algorithm. Unfortunately, this leads to data structure variations during pivoting and unpredictable dependency structures and processor loads. Note that dynamic scheduling has high run-time overhead compared to relatively fine grain computation and that cache performance is usually low. Therefore, parallelizing sparse *LU* factorization is an open problem. Nevertheless, the following two recent packages have achieve promising parallel performance. They both come in two versions: a shared memory and a distributed memory.

SPOOLES implements multithreaded shared memory and MPI-based distributed memory parallel *LU* and *QR* factorization methods. Three ordering schemes are available while sophisticated object-oriented data structures and manipulators are extensively used. Each object has several methods to enter data into, extract data from, and manipulate the data in the objects. It is the first (and probably the only) parallel sparse solver for full-rank overdetermined systems.

Super*LU* implements Gauss elimination with partial pivoting and is particularly appropriate for very unsymmetric matrices. It utilizes the supernode technique and involves both a user-supplied array organized as a two-ended stack (one for the *L* and one for the *U* matrix) and dynamically growing arrays. Although its original version was sequential, carefully designed to exploit memory hierarchies, parallel versions for both shared and distributed memory systems are currently available. In the shared memory implementation an asynchronous scheduling algorithm is used by a scheduler routine which forces a priority-based scheduling policy. This results in significant space and time improvements. A nontrivial modification of super*LU* (57) that is based on a run-time partitioning and scheduling library consists of a distributed memory *LU* implementation that achieved promising parallel efficiency. We should note that for brevity reasons the very recent era of automatic parallelization of sparse computations through parallelizing compilers and run-time compilation techniques (57) is not considered here.

Iterative Solvers

Iterative methods can in principle be parallelized easier than direct methods mainly because they exploit sparseness in a natural way and do not involve fill-ins. Nevertheless, the fundamental trade-off between parallelism (which prefers locality) and fast convergence rate (which is based on global dependence) poses challenges. Next we present some of the recent parallel iterative packages that are publicly available, but the reader is referred to Ref. 58 for more technical details.

AZTEC is a software package that contains several parallel Krylov iterative methods (CG, GMRES, CGS, TFQMR, BiCGstab) to solve general sparse matrices arbitrarily distributed among processors. Its major kernel operation is a parallel sparse matrix–vector one where the matrix and the vectors must be distributed across the processors. In particular the vector elements on each node are ordered into internal, border and external sets depending on the information needed by the matrix–vector kernel. The internal and border elements are updated by the node they are assigned to, and the values of the external elements are obtained via communication whenever the product is performed. Two distributed variants of modified sparse row and variable block row (see SPARSEKIT) data structures are used to store the local submatrices which are reordered in a manner similar to the one used for the local vectors. An additional high-level data interface provides a local to global numbering which facilitates proper communication and synchronization.

A technique similar to the above parallelization approach has been used in //ITPACK, which consists of seven modules implementing SOR, Jacobi-CG, Jacobi-SI, RSCG, RSSI, SSOR-CG, and SSOR-SI under different indexing schemes, and it is integrated in the //ELLPACK system. The code is based on the sequential version of ITPACK which was parallelized by utilizing a subset of sparse BLAS routines.

BlockSolve95 is another iterative library based on parallel BLAS kernels. Its main unique feature is the utilization of powerful heuristic matrix ordering algorithms that allow the usage of higher-level dense BLAS operations. It assumes that a matrix is symmetric in structure and contains CG, SYMMLQ, and GMRES synchronous and asynchronous methods. It exhibits increased parallel efficiency for linear systems that are associated with PDE problems which involve multiple degrees of freedom per node. This is achieved by taking advantage of repeated communication patterns.

Another preconditioned CG-based software system for solving systems of sparse linear algebraic equations methods on a variety of computer architectures is PCG. This software is designed to give high performance with nearly identical user interface across different scalar, vector, and parallel platforms as well as across different programming models such as shared memory, data parallel, and message passing programming interfaces. This portability is achieved through the m4 macro preprocessor. PCG has several levels of access allowing the user to either (1) call the driver routine using one of the predefined sparse matrix data structures as a black box or (2) call at the level of the iterative method with a direct communication interface and user-defined sparse matrix data structures and matrix vector routines or finally use an inverse communication layer that allows full user control and permits more general matrix–vector product operations.

A similar to PCG package that solely focuses on iterative methods is PIM. Its is largely independent of data structures and communication protocols, and the user is expected to customize the basic matrix–vector and vector–vector operations needed in the algorithm on the intended targeting parallel environment.

Finally, P-SPARSLIB follows the methodology of //ITPACK and contains a large class of iterative methods and preconditioners and preprocessing tools. The lower level of this library is a collection of message-passing tools coupled with local BLAS-1 routines. This message-passing toolkit consists of boundary information exchange routines, distributed dot product, send/receive routines used at the preprocessing stage, and auxiliary functions for machine configuration and task synchronization. The rest of the modules, which exploit issues in domain decomposition, distribute sparse data structures, and avoid communication redundancies due to repeated data exchange patterns, are completely machine-independent.

The source code of //ITPACK, PCG, PIM, and P-SPARSLIB is mainly in dialects of Fortran, while AZTEC and BlockSolve95 are written in C/C++. All of the dialects include a PVM and/or an MPI version. It is worth noting that all the above software packages do not partition the matrix across processors. Their performance strongly depends on the assumed partition usually obtained by the algorithms and infrastructure presented in the section entitled "Parallel Dense Linear Algebraic Solvers."

Preconditioning

The most efficient algorithms for solving sparse linear systems appear to be the ones that combine both direct and iterative methods through a technique known as preconditioning. Given the linear system $Ax = b$, a preconditioner matrix M is sought such that it consists of a good approximation of A and is easily invertible. Given such a preconditioner, we solve the preconditioned system $M^{-1}Ax = M^{-1}b$ by standard iterative methods in which only actions on A and M^{-1} are needed. Preconditioners can be either algebraic in nature (constructed solely from the linear system) or PDE-based (using knowledge from the associated continuous PDE problem). Due to its diversity, the latter case will not be considered here. Incomplete factorization preconditioners (ILU class) or iterative matrix-based preconditioners (Jacobi, SSOR, etc.) belong in the first category and are relatively easy to parallelize following the techniques presented previously in this article. Nevertheless, the trade-off between parallelism and fast convergence rate mentioned previously is much more crucial in preconditioning and makes its parallelization very challenging. Assuming that the linear system comes from a PDE discretization method, one can exploit parallelism by using (1) node reordering or grouping, (2) series expansion [Neumann or Euler expansions of $(I - L)^{-1}$, polynomial preconditioners, etc.], (3) domain decomposition (e.g., Schwartz splitting), and (4) multilevel techniques. Most of the iterative packages mentioned above provide at least one (usually Block Jacobi) preconditioner.

There exist two software systems that construct and apply preconditioners in parallel. Both are fully algebraic in the sense that they only need the partitioned matrix and its connectivity information. SPAI provides an approximation L of A^{-1} by considering the minimization problem $\min\|LA - I\|$. The parallelism is due to the fact that this minimization problem is reduced to a set of independent subproblems. Another similar library is ParPre. It focuses on parallel preconditioning based on multilevel (algebraic multigrid) and domain decomposition methods (Schur complement, Additive and Multiplicative Schwartz, and generalized Block Jacobi and SSOR). In addition, it provides a mechanism that allows the user to control the sequentiality while trying to make the right trade-off between parallelism and fast convergence rate. We note that although the BlockSolve95 package contains several iterative methods, it was designed for its multicoloring, fully parallel, incomplete factorization preconditioner.

ACKNOWLEDGMENTS

We are grateful to V. Verykios for the constructive discussion which helped us review the domain decomposition software packages. The work of E. Houstis was supported by the National Science Foundation under grants NSF CDA-9123502 and CCR-92022536 and ARPA grant DAAH04-94-G-0010. The work of A. Sameh and E. Gallopoulos was supported by the National Science Foundation under grant NSF CCR-912010. The work of E. Vavalis was supported by PENED grants 95-602 and 95-107. The work of T. Papatheodorou was supported by ESPRIT Basic Research Action Project APPARC.

BIBLIOGRAPHY

1. T. Chan and R. S. Tuminaro, A survey of parallel multigrid algorithms, *Amer. Soc. Mech. Eng.,* **AMD-86**, 1987.
2. J. Flaherty et al., Parallel Computation in Adaptive Finite Element Analysis, in C. Brebbia and M. Aliabadi (eds.), *Adaptive Finite Element and Boundary Element Methods,* London: Elsevier Applied Science, 1993.
3. M. Pinto, C. Rafferty, and R. Dutton, Pisces-ii Poisson and continuity equation solver, Stanford Univ. Electron. Laboratory Tech. Rep., 1984.
4. E. N. Houstis et al., ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD machines, in D. Marinescu and R. Frost (eds.), *Int. Conf. Supercomput.,* New York: ACM Press, 1990, pp. 96–107.
5. C. Pommerell, M. Annaratone, and W. Fichtner, A set of new mapping and coloring heuristics for distributed-memory parallel processors, *SIAM J. Sci. Stat. Comput.,* **13**: 194–226, 1992.
6. C. Farhat and H. D. Simon, TOP/DOMDEC—a software tool for mesh partitioning and parallel processing, Tech. Rep. RNR-93-011, NASA Ames Res. Center, 1993, pp. 1–28.
7. D. S. Dodson, R. G. Grimes, and J. G. Lewis, Sparse extensions to the Fortran basic linear algebra subprograms, *ACM Trans. Math.Softw.,* **17**: 253–263, 1991.
8. E. N. Houstis and J. R. Rice, Parallel ELLPACK: A development and problem solving environment for high performance computing machines, in P. W. Gaffney and E. N. Houstis (eds.), *Programming Environments for High-Level Scientific Problem Solving,* New York: North-Holland, 1992, pp. 229–241.
9. S. Kim, Parallel numerical methods for partial differential equations, Ph.D. thesis, Tech. Rep. CSD-TR-94-090, Comput. Sci., Purdue Univ., 1993.
10. P. Wu and E. N. Houstis, Parallel mesh generation and decomposition, *Comput. Syst. Eng.,* 1994.
11. S. Baase, *Computer Algorithms: Introduction to Design and Analysis,* Reading, MA: Addison-Wesley, 1988, pp. 145–207.
12. P. Sadayappan and F. Ercal, Nearest-neighbor mapping of finite element graphs onto processor meshes, *IEEE Trans. Comput.,* **C-36**: 1408–1424, 1987.
13. A. George and J. W. H. Liu, An implementation of a pseudoperipheral node finder, *ACM Trans. Math. Softw.,* **5**: 284–295, 1979.

14. M. Al-Nasra and D. T. Nguyen, An algorithm for domain decomposition in finite element analysis, *Comput. Struct.,* **39** (3–4): 227–289, 1991.
15. M. Fiedler, A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory, *Czech. Math. J.,* **25**: 619–633, 1975.
16. H. D. Simon, Partitioning of unstructured problems for parallel processing, *Comput. Syst. Eng.,* **2** (2–3): 135–148, 1991.
17. B. Hendrickson and R. Leland, An improved spectral load balancing method, in *Sixth SIAM Conf. Parallel Proc. Sci. Comput.,* 1993, pp. 953–961.
18. S. T. Barnard and H. D. Simon, A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, *Sixth SIAM Conf. Parallel Process. Sci. Comput.,* 1993, pp. 711–718.
19. M. Loriot and L. Fezoui, Mesh-splitting preprocessor, Tech. Rep., Simulog Inc, 1989.
20. R. D. Williams, Performance of dynamic load balancing algorithms for unstructured mesh calculations, *Concurrency: Practice and Experience,* **3** (5): 457–481, 1991.
21. N. P. Chrisochoides, E. N. Houstis, and J. R. Rice, Mapping algorithms and software environments for data parallel PDE iterative solvers, *J. Distrib. Parallel Comput.,* **21**: 75–95, 1994.
22. H. Byun, E. N. Houstis, and S. Kortesis, A workload partitioning strategy for PDE computations by a generalized neural network, *Neural, Parallel and Sci. Comput.,* **1** (2): 209–226, 1993.
23. V. Verykios and E. N. Houstis, Parallel ELLPACK 3-D problem solving environment, Tech. Rep. TR-97-028, Dept. Comput. Sci., Purdue Univ., 1997.
24. J. J. Dongarra et al., An extended set of basic linear algebra subprograms: Model implementation and test programs, *ACM Trans. Math. Softw.,* **14**: 18–32, 1988.
25. J. J. Dongarra et al., A set of level 3 basic linear algebra subprograms, *ACM Trans. Math. Softw.,* **16**: 1–17, 1990.
26. K. Gallivan, R. Plemmons, and A. Sameh, Parallel algorithms for dense linear algebra computations, *SIAM Rev.,* **32**: 54–135, 1990.
27. G. Birkhoff and R. Lynch, *Numerical Solution of Elliptic Problems,* Philadelphia: SIAM, 1984.
28. P. Henrici, *Discrete Variable Methods in Ordinary Differential Equations,* New York: Wiley, 1962.
29. M. Vajteršic, *Algorithms for Elliptic Problems: Efficient Sequential and Parallel Solvers,* Dordrecht: Kluwer, 1993.
30. R. Lynch, J. Rice, and D. Thomas, Direct solution of partial difference equations by tensor product methods, *Numer. Math.,* **6**: 185–199, 1964.
31. C. Van Loan, *Computational Frameworks for the Fast Fourier Transform,* Philadelphia: SIAM, 1992.
32. J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK,* Berlin: Springer-Verlag, 1985.
33. R. Hockney, (r_∞, $n_{1/2}$, $s_{1/2}$) measurements on the 2-CPU Cray X-MP, *Parallel Comput.,* **2** (1): 1–14, 1985.
34. B. Buzbee, A fast Poisson solver amenable to parallel computation, *IEEE Trans. Comput.,* **C-22**: 793–796, 1973.
35. O. McBryan and E. Van De Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Stat. Comput.,* **8**: s227–s287, 1987.
36. U. Schumann, Comments on "A fast computer method for matrix transposing" and application to the solution of Poisson's equation, *IEEE Trans. Comput.,* **C-22**: 542–544, 1973.
37. D. Gannon and J. V. Rosendale, On the impact of communication complexity on the design of parallel numerical algorithms, *IEEE Trans. Comput.,* **C-33**: 1180–1194, 1984.
38. R. Hockney and C. Jesshope, *Parallel Computers,* Bristol, England: Hilger, 1983.
39. P. Swarztrauber, Multiprocessor FFTs, *Parallel Comput.,* **5** (1–2): 197–210, 1987.
40. B. Buzbee, G. Golub, and C. Nielson, On direct methods for solving Poisson's equation, *SIAM J. Numer. Anal.,* **7**: 627–656, 1970.
41. R. A. Sweet, A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension, *SIAM J. Numer. Anal.,* **14**: 1977, pp. 707–720.
42. P. N. Swarztrauber, A direct method for the discrete solution of separable elliptic equations, *SIAM J. Numer. Anal.,* **11**: 1136–1150, 1974.
43. P. N. Swarztrauber and R. A. Sweet, Algorithm 541: Efficient Fortran subprograms for the solution of separable elliptic partial differential equations, *ACM Trans. Math. Softw.,* **5**: 352–364, 1979.
44. O. Buneman, A compact non-iterative Poisson solver, Tech. Rep. 294, Stanford Univ. Inst. Plasma Res., Stanford, CA, 1969.
45. E. Gallopoulos and Y. Saad, Parallel block cyclic reduction algorithm for the fast solution of elliptic equations, *Parallel Comput.,* **10** (2): 143–160, 1989.
46. E. Gallopoulos and A. H. Sameh, Solving elliptic equations on the Cedar multiprocessor, in M. H. Wright (ed.), *Aspects of Computation on Asynchronous Parallel Processors,* Amsterdam: Elsevier/North-Holland, 1989, pp. 1–12.
47. G. N. Frank, Experiments on the Cedar multicluster with parallel block cyclic reduction and an application to domain decomposition methods, Master's thesis, Dept. Comput. Sci., Univ. Illinois at Urbana-Champaign, 1990.
48. E. Gallopoulos and Y. Saad, Some fast elliptic solvers for parallel architectures and their complexities, *Int. J. High Speed Comput.,* **1**: 113–141, 1989.
49. R. Sweet, Vectorization and Parallelization of FISHPAK, in J. Dongarra et al. (eds.), *Proc. 5th SIAM Conf. Parallel Proc. Sci. Comput.,* Philadelphia: SIAM, 1992, pp. 637–642.
50. R. Hockney, A fast direct solution of Poisson's equation using Fourier analysis, *J. Assoc. Comput. Mach.,* **12**: 95–113, 1965.
51. R. W. Hockney, Characterizing computers and optimizing the facr(l) Poisson solver on parallel unicomputers, *IEEE Trans. Comput.,* **C-32**: 933–941, 1983.
52. R. W. Hockney, The $n_{1/2}$ Method of Algorithm Analysis, in B. Engquist and T. Smedsaas (eds.), *PDE Software: Modules, Interfaces and Systems,* Amsterdam: Elsevier/North-Holland, 1984, pp. 429–445.
53. P. N. Swarztrauber, The methods of cyclic reduction, Fourier analysis and the facr algorithm for the discrete solution of Poisson's equation on a rectangle, *SIAM Rev.,* **19**: 490–501, 1977.
54. C. Temperton, On the facr(l) algorithm for the discrete Poisson equation, *J. Comput. Phys.,* **34**: 314–329, 1980.
55. M. T. Heath, Parallel Direct Methods for Sparse Linear Systems, in A. S. D. E. Keyes and V. Venkatakrishnan (eds.), *Parallel Numerical Algorithms,* Boston: Kluwer, 1997, pp. 55–90.
56. I. S. Duff, Sparse numerical linear algebra: Direct methods and preconditioning, Tech. Rep. RAL-TR-96-047, Rutherford Appleton Laboratory, Oxon, UK, 1996.
57. C. Fu, X. Jiao, and T. Yang, Efficient sparse LU factorization with partial pivoting on distributed memory architectures, *IEEE Trans. Parallel Distrib. Comput.,* **9**: 109–125, 1998.
58. V. Eijkhout, Overview of iterative linear system solver packages, [Online] 1997. Available: ftp://math.ucla.edu/pub/eijkhout/papers/packages.ps

E. N. Houstis
A. Sameh
E. Vavalis
Purdue University
E. Gallopoulos
T. S. Papatheodorou
University of Patras