CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 4(6), 481–497 (SEPTEMBER 1992)

Sparse iterative algorithm software for large-scale MIMD machines: an initial discussion and implementation

JOHN N. SHADID
Parallel Computational Sciences Department
Sandia National Laboratories
Albuquerque, NM, USA

RAY S. TUMINARO
Applied and Numerical Mathematics Department
Sandia National Laboratories
Albuquerque, NM, USA

SUMMARY

The parallelization of sophisticated applications has dramatically increased in recent years. As machine capabilities rise, greater emphasis on modeling complex phenomena can be expected. Many of these applications require the solution of large sparse matrix equations which approximate systems of partial differential equations (PDEs). Therefore we consider parallel iterative solvers for large sparse non-symmetric systems and issues related to parallel sparse matrix software. We describe a collection of parallel iterative solvers which use a distributed sparse matrix format that facilitates the interface between specific applications and a variety of Krylov subspace techniques and multigrid methods. These methods have been used to solve a number of linear and non-linear PDE problems on a 1024-processor NCUBE 2 hypercube. Over 1 Gflop sustained computation rates are achieved with many of these solvers, demonstrating that high performance can be attained even when using sparse matrix data structures.

1. INTRODUCTION

In recent years substantial progress has been made in the numerical solution of scientific applications on large-scale MIMD machines. These machines utilize many independent processors, each with their own local memory, to provide substantial performance increases over vector supercomputers. Unfortunately, this performance increase is often attained with considerable cost in both programming effort and software portability. In particular, users must now subdivide computational tasks into pieces suitable for individual processors and then use messages to achieve synchronization and communication. As machine capabilities continue to increase, greater emphasis on modeling complex phenomena can be expected. However, without advancements in application software methods, utilization of this computational power may considerably lag hardware potential. One possibility is to design parallel software that is applicable to a wide range of scientific problems. In this paper we consider iterative solvers for large sparse non-symmetric systems arising from PDE discretizations and, more generally, issues surrounding parallel sparse matrix software. While robust iterative solver packages are still some years away, we do think it is worthwhile to start considering the unique algorithmic and software issues associated with these methods on large-scale parallel MIMD machines. We explore these issues by considering the parallel implementation of two preconditioned Krylov subspace (or conjugate direction) methods, the conjugate gradient squared (CGS) and the generalized minimum residual method (GMRES), and

1040–3108/92/060481–17$05.00
©1992 by John Wiley & Sons, Ltd.

Received 13 May 1991
Revised 24 February 1992

a multigrid algorithm (MG). Central to these solvers is a distributed sparse matrix format that facilitates the interface between specific applications and a variety of iterative techniques. This format is a generalization of sparse matrix formats found on serial and vector machines.

The use of sparse matrix methods on traditional supercomputers has increased substantially over the past few years. This increase is motivated by the existence of library quality packages as well as a desire for generality and portability in application software. The use of sparse matrix methods facilitates the development of library routines for entire problem classes which can often be highly optimized for a particular machine. This for example, is the case with direct sparse matrix factorization (MA27[1], SMPAK[2], SPARSPAK[3]) and iterative solvers for sparse linear systems (PCGPAK[4], CrayPCG[5], NSPCG[6]). Additionally, the existence of utilities for conversion between sparse matrix formats (SPARSKIT[7]) provides some degree of code modularity. Finally, it is possible for computer vendors to provide routines and entire numerical algorithms which are optimized for their specific machines.

Despite a long list of significant advantages, sparse matrix methods have been slow to achieve widespread acceptance. This is primarily because sparse matrix methods have been traditionally associated with increased storage requirements and vector performance degradation when compared to explicit coding for a specific application. However, we believe that the use of sparse matrix techniques on large-scale distributed memory MIMD machines does not suffer these disadvantages, while retaining the advantages outlined in the preceding paragraph. In particular, the performance degradation on present MIMD processors is not nearly as severe as on traditional vector supercomputers. On these machines the increased overhead of the indirect addressing is significant relative to the high computational throughput of the pipelined vector processors. Further, memory limitations are likely to become less significant on large-scale MIMD machines as the memory on these machines can be slower, and therefore less expensive, than the fast memory needed for high-performance vector processors. These methods also promise to minimize two of the major disadvantages commonly associated with large-scale MIMD machines: (1) the inherent difficulty of programming due to the message-passing paradigm and (2) the need to design special algorithms for each application. It is the intention of this study to present an initial discussion and actual implementation results to substantiate these claims.

This paper continues in Section 2 with a brief discussion of the software framework that has been adopted in designing an iterative solver collection. In the following Sections the parallel structures are described. In particular, mapping issues and the representation of vector variables are discussed in Section 3 and the distributed sparse matrix format is presented in Section 4. In Section 5 the iterative solution methods are briefly described and issues relating to their parallel implementation are considered. Parallel performance results for the 1024-node NCUBE 2 hypercube are discussed in Section 6. Finally, in Section 7 we present some concluding remarks.

2. SOFTWARE FRAMEWORK

In this Section we discuss the overall framework and general philosophy behind our iterative solver collection. This solver collection was developed to examine software and algorithmic issues relevant to future iterative packages on large-scale MIMD machines.

Our focus is primarily directed toward software issues associated with preconditioned Krylov subspace methods. In this paper we describe the current implementation which can be used to solve 2-D non-linear coupled PDE systems on rectangular grids. Additionally, we discuss more general ideas that will be used in a finite element version of these solvers. Our intention is to continually modify and expand the existing software to incorporate more sophisticated and general ideas. Thus, in designing the framework the primary goal is to maintain as much modularity and flexibility as possible.

To facilitate modularity, a distributed sparse matrix philosophy is adopted. In particular, the user supplies a matrix set-up routine in order to apply the solver collection to a PDE problem. Since each processor needs only a subset of the original equations, it is usually possible to modify an existing sparse matrix set-up routine to produce a suitable MIMD version. Of course, communication may also be somewhat application-dependent. In particular, communication requirements for local update operations (e.g. matrix–vector multiply and some preconditioning/relaxation operators) depend on the mapping of the computational problem to the parallel architecture. In our framework, a single subroutine is responsible for maintaining information within each processor so that local updates can occur. Currently, this routine supports relatively simple grids. However, if the communication procedure is inadequate for a specific application, it is possible for users to supply their own communication subroutine. Thus, by changing just one routine, it should be possible to use the majority of the solver package even on problems with no local spatial structure.[1]

A schematic diagram showing the major modules in the PDE/sparse matrix solver is shown in Figure 1. Each MIMD processor of the parallel computer runs the code schematically detailed in Figure 1. The control processor, node 0, reads the problem-specific input data and the choice of solver and preconditioning/relaxation to be used. These data are then broadcast to the remaining nodes in the ensemble. At this point each processor calls the specific distributed sparse matrix routine to set up a subset of the linear equations for the entire problem (see Section 4). After definition of the local matrices the selected parallel algorithm begins. The current solvers include two distributed CG-like methods and an MG routine. These solvers use a combination of local sparse matrix–vector operations, local and global preconditioning (relaxation), and local and global communication to solve a specific linear system. As Figure 1 indicates, it is possible to use a variety of combinations of preconditioners/relaxation methods in conjunction with the different solvers. For non-linear problems Newton's method is used to produce a sequence of these linear problems, each generated by a separate call to the distributed sparse matrix set-up routine to assemble the local Jacobian matrix.

3. DISTRIBUTED REPRESENTATION OF VECTOR VARIABLES

In this Section we discuss the parallelization of the solver collection. In particular, we focus on the local representation of vector variables on each processor and the structures necessary to support communication. However, before describing vector representation we briefly discuss decomposition and mapping of vector variables to multiprocessor machines.

[1] Multigrid methods may require some additional coding or may not even apply. See Section 5.

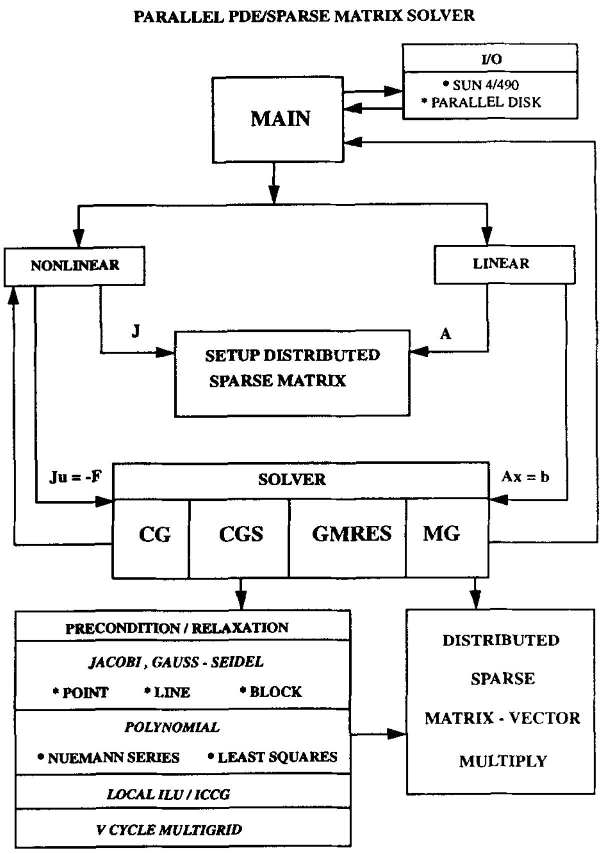

Figure 1. General code organization

Parallelization of the solvers is accomplished by assigning subsets of unknowns to different processors. That is, each processor contains variables which it is responsible for updating. For PDE applications, this partioning usually corresponds to spatially decomposing the computational domain. When finite differences are used, for example, the domain is often partitioned into nearly equally sized rectangles. For more complex grids decomposition techniques have been studied which make use of optimization methods (Fox and Furmanski[8], Williams[9]), graph-based methods (Kernighan and Lin[10]) and heuristic methods (Morrison and Otto[11], Berger and Bokhari[12], Pothen, Simon and Liou[13]). Alternatively, if there is no local structure, then the decomposition can be handled by various strategies which assign $\frac{N}{P}$ unknowns to each processor, where N is the total number of unknowns and P is the number of processors. Once the variables are partitioned into groups, these groups must be mapped to the specific computer architecture. This mapping is generally chosen to minimize the data transfer necessary to complete local operations. That is, the mapping should preserve the computational locality of the problem domain for a specific parallel architecture. For example, in the case of simple one-, two- or three-dimensional rectangular domains a Gray code mapping can be used for hypercube-type architectures (Fox *et. al*[14]) and simple geometric maps with two- or three-dimensional periodic mesh architectures.

Within our solver collection, the user can manually perform the decomposition and mapping of a specific problem by setting the appropriate variables. In addition, we provide automatic routines which decompose and map grids that are logically rectangular. In this case, the decomposition and mapping routines subdivide the domain into rectangular subdomains (of approximately equal size) and then map these subdomains to a hypercube architecture. Extensions necessary for more general mappings (e.g. for finite-element calculations and random sparse matrices) are being incorporated[15].

Given a decomposition and an architecture mapping, we can now consider the distributed representation of vector variables. In particular, each processor must maintain a local numbering of its vector components as well as internal boundary components. These internal boundary components are copies of dependent variables, updated on other processors (called the neighbors), which are necessary to perform all local operations. Within a processor, variables that are updated by this processor are numbered (and thus stored) first, followed by the internal boundary components from each neighboring processor contiguously ordered. By using contiguous sections of memory for both local variables and internal boundary variables, no buffering and scattering of data back to non-contiguous internal boundary structures is necessary when receiving messages. Additionally, this memory arrangement leads to easily optimized kernel operations such as standard vector operations and the local matrix–vector multiply.

In our current implementation we automatically establish the vector representation described above for rectangular subdomains. This is depicted in Figure 2. Specifically, Figure 2(a) represents the decomposition of a regular two-dimensional computational region into subdomains. Each subdomain is then mapped onto the computer architecture. Figure 2(b),(c) depicts the local numbering of vector components in a typical processor. This processor is responsible for updating all dependent variables associated with grid points interior to the interprocessor boundaries.[2] The $\tilde{N}\,(=\frac{N}{P})$ interior points are contained in the set Ω, and are numbered first as shown in Figure 2(c). After these points the interior boundary components contained in the sets $\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4$ are stored in contiguous sections in the vector of dependent variables $\tilde{\mathbf{x}}$. In Figure 2 it is assumed that only an internal boundary depth of one is necessary. A simple modification of this scheme would allow greater depths for higher-order local approximations.

A local communication routine is responsible for updating the points in the Γ_is. In the general case, each processor must contain:

- the number of neighbors
- the processor identification of each neighbor
- the number of components to be received from each neighbor and pointers to where they are stored in $\tilde{\mathbf{x}}$
- the number of components to be sent to each neighbor and pointers to where they are stored in $\tilde{\mathbf{x}}$.

[2] In this discussion we assume that there is only one dependent variable per grid point. However, in the actual implementation we allow multiple dependent variables per grid point and order them consecutively at each grid point.

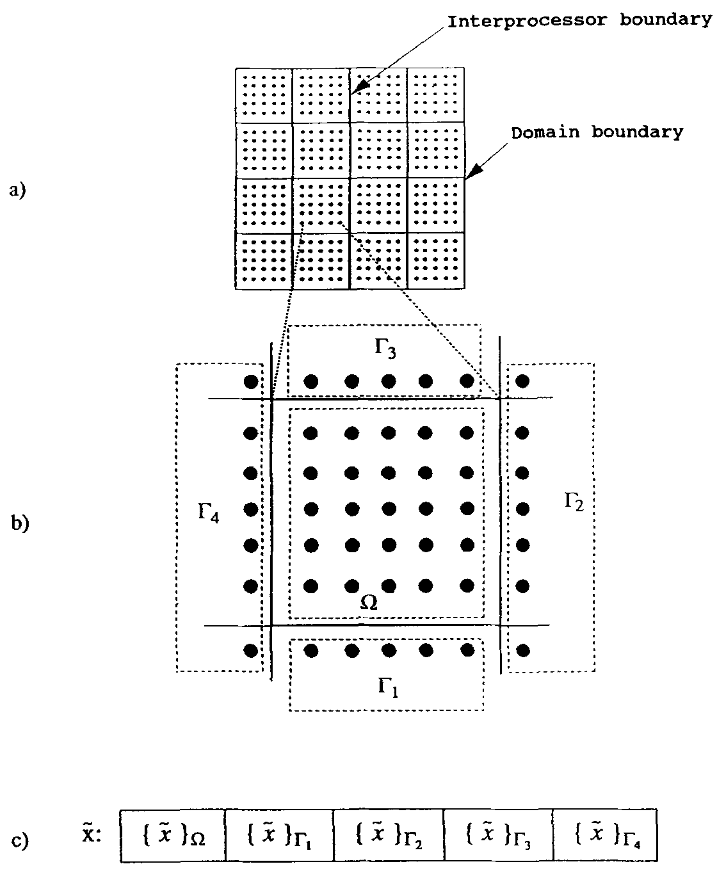

Figure 2. Schematic diagram of local numbering scheme: (a) computational domain; (b) points updated in processor and internal boundary points; (c) vector of dependent variables

In the case of rectangular subdomains this is accomplished using an array *neighbors*[] containing the processor numbers for the four nearest neighbors. In addition, a mapping function, *loc*(), is defined on each processor. This function is used to map the global numbering of dependent variables to a specific local numbering scheme. This map acts on all dependent variables as well as internal boundary components. It can be used to determine the location of components which must be communicated with other processors. For example, in rectangular subdomains the storage location of the corners in $\tilde{x}$ determine the positions of the internal boundaries in $\tilde{x}$. The *loc*() function is also necessary for schemes which use the grid structure (such as multigrid, line or block schemes) so as to map between grid co-ordinates and the local vector representation.

Using these structures, a general communication routine has been implemented for the simple case when the domain of interest has been subdivided into rectangles. Extensions of the structures and the communication routine for more general situations are being incorporated. If the supplied communication procedure is not adequate for a particular application, it is always possible to provide an application-specific routine and use it in conjunction with the iterative solvers. In fact, if necessary, this communication subroutine can involve global transfers for problems which do not rely on local structure.

4. A SPARSE DISTRIBUTED MATRIX FORMAT

Sparse matrix formats allow the structure and non-zero entries of the coefficient matrix A, to be stored while excluding the unnecessary storage of zero entries. In general these methods use one real-valued vector for storage of the non-zeros and one or two integer-valued vectors to describe their location. The various sparse formats differ with respect to their definition of these one-dimensional vectors. The choice of sparse format is often determined by the structure of the matrix A, the way A is used by a specific algorithm, or by performance considerations for a specific computer architecture. A review of sparse matrix formats commonly used on scalar and vector machines is beyond the scope of this paper (see Saad[7]). In the following discussion we present a specific distributed sparse matrix representation. This format is a generalization of a sequential sparse matrix format, the modified sparse row (MSR) format, to the distributed memory MIMD environment.

The data structure for the MSR format consists of a real array **a** and an integer array **ja**. Symbolically we associate the matrix A with the vector pair (**a**,**ja**). The first N positions in **a** contain the diagonal elements of the matrix, in order. The position $N+1$ of the array **a** is unused. The first $N+1$ positions of **ja** contain a pointer to the beginning of each row in **a** and **ja**. These entries are also used to determine the number of offdiagonal non-zeros per row in A. Starting from position $N+2$, the non-zero offdiagonal elements of A are stored row-wise. Corresponding to each offdiagonal element $\mathbf{a}(k)$ the integer $\mathbf{ja}(k)$ is the column index of the element $\mathbf{a}(k)$ in the matrix A. The advantage of this format is that a number of simple scaling techniques, preconditioners and relaxation methods use these diagonal terms directly. Therefore a format in which they are stored contiguously and readily available (without searching or preprocessing) is desirable.

Using the sequential MSR format the distributed modified sparse row (DMSR) format is defined as follows. For clarity of presentation, let us assume that $\frac{N}{P}$ variables have been assigned to each of the P processors as discussed in Section 3. Corresponding to this variable partioning, we decompose the matrix A into P submatrices $\bar{\mathbf{A}}_i$, where each submatrix corresponds to $\frac{N}{P}$ rows of the matrix A. Each submatrix is then assigned to one of the processors such that all variables (those updated by the processor as well as the internal boundary components) corresponding to each equation are stored locally within that processor. Each rectangular submatrix $\bar{\mathbf{A}}_i$, is then represented by an MSR format sparse matrix $\bar{\mathbf{a}}_i$, $\bar{\mathbf{j}}\mathbf{a}_i$) with a local numbering of the dependent variables and the corresponding equations as discussed above. Obviously, in the limit of one processor, the DMSR format reduces to the normal sequential MSR format. Further, no additional structures are necessary to link the local submatrices. That is, the structures described in the preceding Section for linking the vector components, *loc*() and *neighbors*(), implicitly link the distributed sparse matrix.

In defining the distributed sparse matrix format almost any row-oriented format (MSR, CSR, ELL, JAD[7]) could have been used as a basis. Another important point is that a number of utilities and format conversion routines from SPARSKIT can be used with the new format. This is because each of the local matrices ($\bar{\mathbf{a}}_i$, $\bar{\mathbf{j}}\mathbf{a}_i$) in the DMSR format conform to the MSR standard. Thus, code portability of kernel algorithms such as preconditioners and relaxation routines is facilitated. Additionally, a substantial amount of code can be reused from the sequential environment.

5. ITERATIVE SOLVERS

For iterative solvers, we consider both Krylov subspace and multigrid methods applied to the linear system

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}$$

These algorithms are among the fastest iterative solvers for a wide variety of applications (e.g. CFD). In our work, we focus primarily on Krylov methods even though they are usually slower than multigrid methods. This is because they are, in general, more broadly applicable to a wide range of sparse matrix problems than multigrid methods. In particular, suitable multigrid operators (e.g. interpolation, projection, coarse grid difference equations, relaxation) cannot be entirely separated from the PDE application being solved. Further, multigrid methods require knowledge of the underlying grid and can be cumbersome to program for complex geometries.

The most well known of the Krylov algorithms is the conjugate gradient method (CG) originally proposed by Hestenes and Steifel[16]. The CG algorithm has the property that the residual is minimized over the space of solutions given by

$$x = x_0 + v \quad , \quad v \in K_m$$

where K_m is the Krylov subspace defined as

$$K_m = \{r_0, \mathrm{A}r_0, \mathrm{A}^2 r_0, \ldots, \mathrm{A}^{m-1} r_0\}$$

r_0 is the initial residual and m is the number of conjugate gradient steps taken. A key factor in the practical implementation of the CG method is that the new iterates can be computed by a simple, inexpensive recurrence relation. Unfortunately, the combination of the residual minimization property along with the simple recurrences of the CG algorithm rely on the matrix being symmetric positive-definite. To handle non-symmetric systems a number of related algorithms have been proposed. For the most part, these fall into two categories. One approach used by BiCG, CGS, and Bi-CGSTAB (see Sonneveld[17] and Van der Vorst[18]) is to modify the computations so that simple recurrence relations can still be used to generate the search directions. This is done by enforcing slightly different orthogonality relations while sacrificing the residual minimization property. Another approach used by GMRES (see Saad[19]) is to maintain the minimization property but give up the recurrence relations. In particular, GMRES projects the original equations onto the Krylov subspace and solves the resulting least-squares problem. Since each direction vector needs to be saved, the storage requirements grow as the iteration proceeds. Thus, the algorithm is normally restarted to avoid excessive growth in storage. Typically, Krylov methods are used in conjunction with a preconditioner, $\mathbf{M}$, whose inverse is an inexpensive approximation to A^{-1}. For the most part, the overall performance of the iterative method depends heavily on the preconditioner. Some popular preconditioners include: polynomial, Jacobi, Gauss–Seidel, SSOR, ILU and domain-decomposition techniques. The CGS and GMRES method are listed in Figures 3 and 4.

From a software optimization perspective, one important advantage of the Krylov methods is that only a few simple kernels are used. For the CGS algorithm these

```
CGS($x_0$ , $A$ , $b$ )
    $r_0 := b - Ax_0$ ;   $q_0, p_{-1} := 0$ ;
    $\tilde{r}_0 = r_0$ ;   $\rho_{-1} := 1$ ; $n := 0$ ;
    while ( residual > tolerance ) do
        $\rho_n := \tilde{r}_0^T r_n$ ;   $\beta := \frac{\rho_n}{\rho_{n-1}}$;
        $u := r_n + \beta q_n$;
        $p_n := u + \beta(q_n + \beta p_{n-1})$;
        $\hat{p} = M^{-1} p_n$ ;   /* $M$ is preconditioning matrix. */
        $v := A\hat{p}$ ;
        $\alpha := \rho_n / (\tilde{r}_0^T v)$ ;
        $q_{n+1} := u - \alpha v$ ;
        $\hat{u} = M^{-1}(u + q_{n+1})$ ;
        $r_{n+1} := r_n - \alpha A\hat{u}$ ;
        $x_{n+1} := x_n + \alpha\hat{u}$ ;
        $n := n + 1$ ;
    end
end
```

Figure 3. CGS algorithm

```
GMRES($x_0$,$m$,$A$,$b$ )
    while( not converged ) do
        /* Arnoldi process */
        $r_0 := M^{-1}(b - Ax_0)$ ;            /* $M$ is preconditioning matrix */
        $\beta = \sqrt{(r_0^T r_0)}$ ;   $v_1 := r_0/\beta$ ;
        for $j = 1$ , ... , $m$   do
            for $i = 1,...,j$   do   $h_{ij} := v_i^T M^{-1} A v_j$ ;   end
            $\hat{v}_{j+1} := M^{-1}Av_j - \sum_{i=1}^{j} h_{ij} v_i$ ;
            $h_{j+1j} := \sqrt{(\hat{v}_{j+1}^T \hat{v}_{j+1})}$ ;   $v_{j+1} := \frac{\hat{v}_{j+1}}{h_{j+1j}}$ ;
        end
        Let $y_m$ be the value of $y$ minimizing $||\beta e_1 - Hy||_2$ ;
                                          /* $H$ is $(m+1) \times m$ upper Hessenberg
                                             matrix with non-zeros given by $h_{ij}$
                                             and $e_1 = [1,0,...,0]^T$. */
        $x_0 := x_0 + V_m y_m$                 /* $V_m = [v_1,...,v_m]$. */
    end
```

Figure 4. Restarted GMRES algorithm (Gram–Schmidt orthogonalization)

include dot products, vector addition and matrix–vector multiplication. In addition to these kernels, GMRES requires a Gram–Schmidt process and a plane rotation algorithm for the least-squares computation. Both of these can be built on top of the simpler kernels.

Notice that most of the kernels involve BLAS operations (basic linear algebra subroutines). These correspond to simple vector manipulations such as dot products, vector addition and vector–scalar multiplication. Of these standard routines, only the dot products require any global communication. In particular, one needs to sum local dot product information from each processor. This can be accomplished efficiently using $2\log_2(P)$ messages.[3] Since these BLAS routines are fairly standard, it is possible that they can be supplied (and optimized) by the vendor. The only other kernels required besides the BLAS routines are the matrix–vector multiply and the preconditioning steps. In most cases, the matrix–vector multiplication will only involve local communication (due to the locality of most PDE approximations). Further, many of the preconditioners (e.g. polynomial, Gauss–Seidel, Jacobi) also only involve local communication. Finally, it is important to notice that only the matrix–vector multiplication and (perhaps) the preconditioning step actually involve the specific application. This simplifies software development as the basic algorithm and many preconditioners can be designed with only limited knowledge of the discrete equations.

In addition to Krylov schemes, we consider multigrid methods (MG) (see Reference 21 for introductory material on multigrid methods). One iteration of a simple multigrid 'V' cycle consists of smoothing the error using a relaxation technique, 'solving' an approximation to the smooth error equation on a coarse grid, interpolating the error correction to the fine grid, and finally adding the error correction into the approximation. An important aspect of the multigrid method is that the coarse grid solution can be approximated by recursively using the multigrid idea. That is, on the coarse grid, relaxation is performed to reduce high-frequency errors followed by the projection of a correction equation on yet a coarser grid, and so on. Thus, the MG method requires a series of problems to be 'solved' on a hierarchy of grids with different mesh sizes. We summarize this procedure in Figure 5.

```
MG(b,u,level)
  if ($level$ == CoarsestLevel ) then $u = A_{level}^{-1}b$
  else
    $u$ :=relax($b,u,level$)
    $r := b - Au$
    $\bar{r} := Rr$          /* $R$ is a projection operator */
    $v := 0$
    MG($\bar{r},v,level + 1$)
    $u := u + Pv$        /* $P$ is an interpolation operator */
  endif
```

Figure 5. MG algorithm

[3] It is possible to reduce the overall communication required when many independent dot products are to be computed. In particular, the communication can be grouped so that only 2 log P messages are needed to sum all the dot products. See Reference 20 for algorithmic/communication issues surrounding Gram–Schmidt vs. modified Gram–Schmidt.

In general, multigrid algorithms require somewhat higher-level kernels than Krylov methods. The principle kernels are interpolation, projection and relaxation. One difficulty is that the exact form of these operators depends on the specific application. Nonetheless, it is possible to write kernel routines that are suitable for large problem classes. For example, interpolation and projection kernels can be defined which assume a particular structure to the grids as well as the order of the differential equation. These routines can then be optimized for high performance similar to the Krylov kernels. One important aspect concerning multigrid methods is that all of its kernels are highly parallel in nature. In particular, it is usually possible to perform updates required at all points within a relaxation method or a grid transfer operator in parallel. Further, only local communication is required for interpolation, projection and most relaxation schemes (such as distributed Gauss–Seidel).[4]

For both Krylov and multigrid methods, there are a number of theoretical convergence results putting them on a sound mathematical basis. Most of the Krylov methods are guaranteed to converge in at least N iterations in the absence of rounding error, where N is the number of unknowns. While this is somewhat slow, these methods converge in far fewer than N iterations when applied to many problems. For multigrid methods, the convergence results generally make assumptions on the ellipticity of the operator and the quality of the coarse-grid operators in approximating the PDE on large scales. The principal result is that, under suitable conditions, the convergence rate is independent of the mesh spacing. The actual algorithm performance is closely tied to the preconditioner in the Krylov techniques and the relaxation scheme in the multigrid method. The overall effectiveness of these operators is somewhat problem-dependent. Thus, any kind of real package must allow for users to write their own preconditioners and relaxation schemes as well as handle convergence issues. These issues, however, are not unique to the parallel environment. For this reason the recent work on serial and vector machines ([6] and [23]) will be applicable.

Finally, we discuss some additional software issues involved with multigrid methods. In general, multigrid methods when applicable are the fastest solvers, often producing solutions at substantial savings over the Krylov methods. In general, the reason for this exceptional performance is that the multigrid algorithm takes advantage of the fact that it is solving an approximation to a continuous PDE operator. In particular, the coarse grid discretizations are an attempt to capture the PDE behavior on widely varying spatial scales. To do this properly, a detailed analysis of the PDE problem (and the corresponding discretizations) is often required. For example, boundary conditions, discontinuities and singularities as well as discretization stability (on all grid levels) should be addressed carefully. Thus, when using a simple multigrid implementation one can expect significant performance degradation over a multigrid procedure which has been more thoughtfully developed. A second issue in writing multigrid software is that the grid structure must be incorporated into the algorithm. That is, one needs to define a series of grids and the corresponding transfer operations between them. When the geometry is complicated, this can require some fairly tedious programming.

In our current implementation, we handle these issues in the simplest fashion. In particular, we assume that the grid is rectangular, using two arrays to specify the grid

[4] Non-local communication may be required for computations on very coarse grids (when there are fewer grid points than processors). This is not the case on hypercube computers when a suitable mapping is used[22].

spacing in the x- and y-directions. Further, we assume a certain ordering of the equations in the PDE matrix and provide routines which map from the matrix ordering to the grid points. The algorithm calls the matrix set-up routine (supplied by the user) once for each grid that needs a discretization. Bilinear interpolation, as well as full weighting restriction, are used for grid transfer operators. In addition, a number of basic relaxation schemes have been provided, including Jacobi, Gauss–Seidel and line Gauss–Seidel.

6. PERFORMANCE RESULTS

To test the flexibility, robustness and numerical properties of the algorithms, a number of test problems were solved on the NCUBE 2. In this paper we present the results from two PDE problems. The first is a finite-difference approximation (using central differences) to a convection–diffusion linear PDE taken from Van der Vorst[24]:

$$-\frac{\partial^2 u}{\partial x^2}+\frac{\partial u}{\partial x}+(1+y^2)(-\frac{\partial^2 u}{\partial y^2}+\frac{\partial u}{\partial y})=f(x,y) \qquad (1)$$

defined on the unit square with Dirichlet boundary conditions and a right-hand side such that

$$u(x,y)=e^{x+y}+x^2(1-x)^2\ln(1+y^2) \qquad (2)$$

The second PDE corresponds to the incompressible Navier–Stokes and thermal energy equations describing buoyancy induced natural convection in an inclined two-dimensional rectangular cavity over the computational domain $(-1,1)\times(-1,1)$. The equations given in non-dimensional, stream-vorticity formulation are:

$$\frac{\partial^2\Omega}{\partial x^2}+A_x{}^2\frac{\partial^2\Omega}{\partial y^2}=\Pi_1\frac{\partial\Phi}{\partial x}+\Pi_2\frac{\partial\Phi}{\partial y}$$

$$\frac{\partial^2\Psi}{\partial x^2}+A_x{}^2\frac{\partial^2\Psi}{\partial y^2}=\frac{A_x{}^2}{4}\Omega$$

$$A_x\frac{\partial\Psi}{\partial y}\frac{\partial\Phi}{\partial x}-A_x\frac{\partial\Psi}{\partial x}\frac{\partial\Phi}{\partial y}=\frac{\partial^2\Phi}{\partial x^2}+A_x{}^2\frac{\partial^2\Phi}{\partial y^2}$$

where

$$\Pi_1\equiv -A_x\frac{Ra\cos\Theta}{2}, \qquad \Pi_2\equiv A_x{}^2\frac{Ra\sin\Theta}{2} \qquad (3)$$

A_x is the aspect ratio of the cavity, Θ is the angle of inclination and Ra is the Raleigh number. In our example problem, we take $A_x = 1$, $\Theta = \pi/2$ and $Ra = 10^3$. Dirichlet boundary conditions are used for the stream function, Ψ, a mixed Dirichlet/Neumann condition for temperature, Φ, and a first order approximation of the stream function–vorticity relationship at the boundary is used for the vorticity boundary conditions (see Reference 25). A discrete approximation is obtained using central-difference approximations.

Our emphasis in this Section is simply to illustrate that high performance is attainable using a distributed sparse matrix format and that this format enables a great deal of flexibility in solving a variety of problems. We begin by looking at a few of the Krylov method kernels in terms of their individual performance on the NCUBE 2. In Table 1 we illustrate the high megaflop rates attained for these kernels for the two example problems. In Table 1, *sparax* refers to the multiplication of the sparse matrix with a vector, *ddot* corresponds to a vector dot product, *daxpy* is a scalar–vector product followed by a vector add, and *dscal* is a scalar–vector multiplication. The kernel megaflop rates are given for a 1024 x 1024 computational grid. The convection–diffusion problem contains about 1 million unknowns with approximately 5 million non-zeros in the matrix. The natural convection problem contains about 3 million unknowns and has approximately 27 million non-zeros in the corresponding Jacobian matrix. From the Table we can see that the *daxpy* operation is computed at over 2 Gflops. This compares well with the vendor's listed maximum computation rating of 2.5 Gflops for double precision. One reason for this high performance is that the BLAS routines were actually provided by the vendor and are well tuned to the machine. Since the only other basic kernel for the Krylov methods is a short matrix–vector multiply routine, we have written it in assembly code (with the help of the vendor) to attain very high performance. It is interesting to compare the performance of these BLAS routines with that of the sparse matrix–vector multiply. In particular, this routine achieves a computation rate of 1.35 Gflops (approximately a factor of 2 degradation over the vendor's BLAS routines). This corresponds to performing 150 matrix–vector multiplies with a matrix containing over 5 million non-zeros in about 1 s. It is important to realize that this high performance is, in fact, achieved using a sparse-matrix format (requiring more sophisticated addressing).

Run-time results for both the non-linear natural convection problem and the convection–diffusion problem are given in Tables 2 and 3. In both cases, a solution is obtained efficiently and quickly.[5] Note that for the convection–diffusion problem

Table 1. Performance of kernel routines (MFLOPS)

routine\ procs	64	128	256	512	1024
sparax[1]	85[†] (82)[‡]	169 (159)	338 (310)	676 (577)	1348 (1060)
sparax[2]	—	—	286 (279)	573 (547)	1145 (1071)
ddot[1]	131 (121)	262 (220)	521 (366)	1043 (529)	2077 (663)
ddot[2]	—	—	523 (457)	1046 (787)	2090 (1211)
daxpy[1]	134	268	535	1066	2121
dscal[1]	116	231	461	920	1825

[1]Convection-diffusion problem. [2]Natural convection problem.

[†]Terms without parenthesis are based only on computation time.

[‡]Terms inside parenthesis are based on computation and communication time.

—Not enough memory to run kernel performance routine.

[5] The natural convection problem is more difficult than the convection–diffusion equation due to the interaction of the coupled PDEs and the inherent non-linearity of the problem.

Table 2. Run times (s) for 128 × 128 natural convection problem

algorithm\	procs	16	64	128	256	512
CGS:	LS1	712.9	215.5	123.6	77.9	50.1
	NS1	660.4	224.2	114.0	68.4	41.2
	NS5	1096.4	297.2	182.1	103.0	66.3
	mg(rbgs1)	229.9	70.2	43.3	28.5	23.9
GMRES:	mg(rbgs1)	246.9	77.7	48.8	32.3	24.1
	mg(jac3)	362.1	110.2	66.7	43.7	32.1
MG:	rbgs3	136.4	42.1	25.8	17.5	13.2
	jac7	387.7	117.3	71.2	46.8	34.9

Preconditioning/relaxation: jac - Jacobi, rbgs - red black Gauss–Seidel, NSn = *n*-term Neumann series polynomial, LSn = *n*-term least squares polynomial, mg(relaxn) = V cycle multigrid with *n* pre- and postrelaxation sweeps using relax. GMRES method uses a Krylov subspace size of 25 before restarting

Table 3. Performance of iterative solvers for the convection–diffusion problem

algorithm\ procs		256 (512 × 512 grid)			1024 (1024 × 1024 grid)		
		Time (s)	Speed-up	Mflops	Time (s)	Speed-up	Mflops
CGS:	jac	57.6	205	284	130.7	802	1110
	rbgs	78.0	230	140	131.1	890	550
	LS3	35.0	215	272	70.8	860	1080
	NS1	41.7	220	252	86.1	870	1000
	mg(rbgs1)	4.7	202	65	4.9	773	249
	mg(jac1)	7.6	203	92	7.8	792	357
GMRES:	jac	578.5	236	457	2123.6	930	1800
	rbgs	257.7	240	344	885.2	952	1365
	LS9	61.0	248	350	184.8	993	1400
	NS11	129.7	241	323	424.4	961	1289
	mg(rbgs1)	6.5	201	82	6.7	779	318
	mg(jac1)	8.1	210	119	8.3	818	464
MG:	rbgs2	3.0	189	52	3.1	732	200
	jac4	4.9	188	119	5.1	723	459

GMRES method uses a Krylov subspace size of 64 before restarting

extremely high megaflop rates are attained by the Krylov solvers (not including those using multigrid preconditioning) on the NCUBE 2 for the 1024 × 1024 grid. This is not surprising in that the kernel operations discussed above comprise almost the entire computation of a Krylov iteration. The multigrid megaflop rates are somewhat lower. This is primarily due to the fact that the multigrid kernels were not optimized to the same degree as the Krylov routines. This optimization was not performed for a number of reasons. In particular, there are more multigrid kernels to optimize, these kernels are more difficult to optimize; and they can only be applied to a somewhat limited range of problems. By contrast, the Krylov methods required only one fairly general purpose subroutine, *sparax*, to be optimized. In addition to megaflop rates, Table 3 illustrates that the scaled speed-ups[6] for all the solvers on the convection–diffusion problem are quite high. That is, these routines make good usage of all the processors for large problems. Note that the multigrid schemes are the fastest in overall run time. This is dramatically apparent on the very large grids (e.g. 1024 x 1024) where the mesh spacing is small. On grids with larger mesh spacing (as one might have for 3-D calculations), the multigrid methods are still the fastest, but the relative performance improvement over the Krylov schemes is not as great. Unfortunately, the class of problems for which we can apply this multigrid algorithm is smaller than those which the Krylov methods apply due to limitations on the grid and finite-difference operator. Finally, we reiterate that the Tables given in this paper are not intended to be used as a precise comparison of the different methods. Our only aim in this paper is to illustrate that we are able to solve a variety of PDE problems with a parallel solver collection and that high megaflop rates and very fast solution times are often possible. We defer a detailed comparison of the individual solvers and preconditioners to other studies ([20,27]).

7. CONCLUSION

In this study we have discussed a collection of iterative routines based on a general-purpose distributed sparse matrix format. In particular, we have motivated the use of such a structure and contrasted its use on parallel machines with the more traditional vector application. This structure allows the design of high-performance parallel iterative methods which are relatively application-independent. To verify these claims we have presented an initial implementation of a variety of methods on a 1024-processor NCUBE 2 hypercube. The majority of these solvers are based on Krylov subspace methods. These techniques are built almost exclusively on top of BLAS routines supplied by the vendor and a distributed sparse matrix multiplication routine. Owing to the relative simplicity of these kernels, it was possible to optimize the majority of the Krylov method code. This optimization allowed over 1 Gflop performance to be attained with many of the solvers. We believe that the most important aspect of this work is the combination of high computational performance with the relatively application-independent nature of the parallel preconditioned Krylov methods. On large-scale MIMD machines, the ability to solve a wide variety of problems without requiring significant recoding is an obvious major advantage.

ACKNOWLEDGEMENTS

We should like to thank M.E. Lewitt of NCUBE for assistance in writing the assembly code version of the sparse matrix–vector multiply. This work was supported by the Applied Mathematical Science Program, US Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories operated for the US Department of Energy under contract no. DE-AC04-76DP00789.

[6] We define scaled speed-ups by the ratio $(p\,T_1)\,/\,T_p$, where p is the number of processors and T_p is the time required to run one iteration of a problem with a total of $p * n$ unknowns using p processors. See Reference 26 for a general discussion of scaled speed-ups

REFERENCES

1. I.S. Duff and J.K. Reid, *MA27—a Set of Fortran Subroutines for Solving Sparse Symmetric Sets of Linear Equations*, Tech. Report AERE R-10533, Computer Science and Systems Division, AERE Harwell, Oxfordshire, England, 1982.
2. S.C. Eisenstat, M.C. Gursky, M.H. Schultz and A. Sherman, *Yale Sparse Matrix Package. I: Symmetric Problems*, Tech. Rep. 112, Department of Computer Science, Yale University, New Haven, CT, 1977.
3. A. George and J. Liu, 'The design of a user interface for a sparse matrix package', *ACM Trans. Math. Soft.*, **5**, 139–162 (1979).
4. *PCGPAK/Cray and PCGPAK/FPS benchmarks*, Tech. Rep. SCA-111, Scientific Computing Associates, Inc., New Haven, CT, September 1987.
5. M. Heroux, P. Vu and C. Yang, *A Parallel Preconditioned Conjugate Gradient Package for Solving Sparse Systems on a Cray Y-MP*, Tech. Rep., Cray Research, Inc., 1408 Northland Dr., Mendota Heights, MN, 1990.
6. T. Oppe, W. Joubert and D. Kincaid, *NSPCG User's Guide, Version 1.0*, Tech. Rep. CNA-216, Center for Numerical Analysis, University of Texas, Austin, Texas, April 1988.
7. Y. Saad, *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*, Tech. Rep., Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, Apr. 1990.
8. G.C. Fox and W. Furmanski, 'Load balancing loosely synchronous problems in a neural network', in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, G.C. Fox (ed.), ACM Press, New York, 1988, pp. 241–278.
9. R. Williams, *Optimization by a Computational Neural Net*, Tech. Rep. C3P371, Caltech Concurrent Computing Project, Pasadena, CA, Nov. 1986.
10. B. Kernighan and S. Lin, 'An efficient hueristic procedure for partitioning graphs', Tech. Rep., *Bell Syst. Tech. J.*, **49** (1970).
11. R. Morrison and S. Otto, *The Scattered Decomposition for Finite Elements*, Tech. Rep. Memo 286, Caltech Concurrent Computing Project, Pasadena, CA, May 1985.
12. M. Berger and S. Bokhari, 'A partitioning strategy for non-uniform problems on multiprocessors', *IEEE Trans.*, **C-36**, 570–580 (1987).
13. A. Pothen, H.D. Simon and K. Liou, 'Partitioning sparse matrices with eignevectors of graphs', *SIAM J. Matrix Anal. Appl.*, **11**, 430–452 (1990).
14. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
15. S. A. Hutchinson, J. N. Shadid and K. T. Ng, *A Comparison of Sparse Matrix Methods on Distributed Memory SIMD and MIMD machines*, Tech. Rep. to be published, Sandia National Laboratories, Albuquerque, NM, 1991.
16. M. Hestenes and E. Stiefel, 'Methods of conjugate gradients for solving linear systems', *J. Res. NBS*, **49**, 409–435 (1952).
17. P. Sonneveld, 'CGS: A fast Lanczos-type solver for nonsymmetric linear systems', *SIAM J. Sci. Stat. Comput.*, **10**, 36–52 (1989).
18. *Bi-CGSTAB: A Fast and Smoothly Converging Variant of CG-S for the Solution of Non-symmetric Linear Systems*, Tech. Rep., University of Utrecht, Budapestlaan 6, Utrecht, The Netherlands, Oct 1990.
19. Y. Saad and M. Schultz, 'GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems', *SIAM J. Sci. Stat. Comput.*, **7**, 856–869 (1986).
20. J.N. Shadid and R.S. Tuminaro, *A Comparison of Preconditioned Krylov Methods on a Large-Scale MIMD Machine*, Tech. Rep. to be published as Sand91-0333, Sandia National Laboratories, Albuquerque, NM, 1991.
21. W. Briggs, *Multigrid Tutorial*, SIAM, Philadelphia, 1987.
22. T. Chan and Y. Saad, 'Multigrid algorithms on the hypercube multiprocessor', *IEEE Trans.*, **C-35**, 969–977 (1986).
23. S. Ashby and M. Seager, *A Proposed Standard for Iterative Linear Solvers: Version 1.0*, Tech. Rep., Lawrence Livermore Nat. Labs., January 1990.
24. H. Van der Vorst, 'Iterative solution methods for certain sparse linear systems with a

nonsymmetric matrix arising from PDE problems', *J. Comput. Phys.*, **44**, 1–19 (1981).

25. J.N. Shadid, *Experimental and Computational Study of the Stability of Natural Convection Flow in an Inclined Enclosure*, Ph.D. thesis, University of Minnesota, 1989.
26. J. Gustafson, G. Montry and R. Benner, 'Development of parallel methods for a 1024-processor hypercube', *SIAM J. Sci. Stat. Comput.*, **9**, 609–638 (1988).
27. J.N. Shadid and R.S. Tuminaro, 'Iterative methods for nonsymmetric systems on mimd machines', to be published in *Proceedings of Fifth SIAM Conference on Parallel Processing for Scientific Applications*, SIAM, 1991.