PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices

SALVATORE FILIPPONE
IBM Italia
and
MICHELE COLAJANNI
Università di Modena e Reggio Emilia

Many computationally intensive problems in engineering and science give rise to the solution of large, sparse, linear systems of equations. Fast and efficient methods for their solution are very important because these systems usually occur in the innermost loop of the computational scheme. Parallelization is often necessary to achieve an acceptable level of performance. This paper presents the design, implementation, and interface of a library of Basic Linear Algebra Subroutines for sparse matrices (PSBLAS) which is specifically tailored to distributed-memory computers. PSBLAS enables easy, efficient, and portable implementations of parallel iterative solvers for linear systems. The interface keeps in view a Single Program Multiple Data programming model on distributed-memory machines. However, the architecture of the library does not exclude an implementation in different paradigms, such as those based on the shared-memory model.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; D.3.2 [**Programming Languages**]: Language Classifications—*FORTRAN*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse, structured and very large systems* (direct and iterative methods); G.4 [**Mathematics of Computing**]: Mathematical Software—*Parallel and vector implementations*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Basic Linear Algebra Subprograms

1. INTRODUCTION

Many scientific applications that motivate the use of a parallel computer require the solution of large and sparse systems of equations. In this paper

Authors' addresses: S. Filippone, Università di Roma Tor Vergata, Centro di Calcolo e Documentazione, Via Orazio Raimondo 18, Roma, Italy-00173; email: sfilippone@uniroma2.it; M. Colajanni, Dipartimento di Scienze dell'Ingegneria, Università di Modena e Reggio Emilia, Via Campi 213, Modena, Italy-41100; email: colajanni@unimo.it; colajanni@uniroma2.it.

we present the software architecture and some implementation features of a library of Basic Linear Algebra Subroutines for parallel sparse applications, namely Parallel Sparse BLAS (PSBLAS), that facilitates parallelization of computationally intensive scientific applications. The PSBLAS library is designed to address parallel implementation of iterative solvers for sparse linear systems through the distributed-memory paradigm. It includes routines for multiplying sparse matrices by dense matrices, solving block diagonal systems with triangular diagonal entries, preprocessing sparse matrices, and contains additional routines for dense matrix operations. The project has been prompted by the appearance of the proposal for serial sparse BLAS [Duff et al. 1997]. We verified that the routines of the serial sparse library were flexible and powerful enough to be used as the building blocks of more complex applications, such as those underlying the PSBLAS parallel library.

Sparse iterative solvers are the kernels of many software systems covering a broad area of applications, such as the simulation of internal combustion engines, oil reservoirs, semiconductor devices, structural analysis, and electromagnetic scattering; a review of these applications can be found in Barrett et al. [1994]. The most widely used methods are those based on Krylov-subspace approximations; the first and perhaps best known is the Conjugate Gradient method [Kelley 1995; Greenbaum 1997]. Our library should be useful to users interested in complex simulations requiring very large discretization meshes. Our own experience has been mainly in applying the PSBLAS solvers to computational fluid dynamics applications. As a typical case, a complete simulation of an internal-combustion engine can easily scale up to some millions of variables.

The applicability of sparse iterative solvers to many different areas causes some terminology problems because the same concept may be denoted through different names depending on the application area. We will describe our results mainly in terms of finite difference discretizations of Partial Differential Equations (PDEs). However, the scope of the PSBLAS library is wider than that: for example, it can be applied to finite element discretizations of PDEs, and even to different classes of problems such as nonlinear optimization, for example in optimal control problems.

The design of a solver for sparse linear systems is driven by many conflicting objectives, such as limiting occupation of storage resources, exploiting regularities in the input data, exploiting hardware characteristics of the parallel platform. To achieve an optimal communication-to-computation ratio on distributed-memory machines it is essential to keep the *data locality* as high as possible; this can be done through an appropriate data allocation strategy. The choice of the preconditioner is another very important factor that affects efficiency of the implemented application. Optimal data distribution requirements for a given preconditioner may conflict with distribution requirements of the rest of the solver. Finding the optimal trade-off may be very difficult because it is application dependent. In this paper, we focus on preconditioners that act on local data elements or have the same communication requirements as those of the matrix-vector

product. In our experience, we found that these preconditioners achieve a good compromise between computation and communication costs of the application and effectiveness in improving the convergence of the iterative method.

The PSBLAS library is internally implemented in a mixture of Fortran 77 and C language. The interface, built over the top of the Fortran 77 kernels, is based on Fortran 90 [Metcalf and Reid 1990]. This language choice for the interface is at the right level of abstraction for the target applications of the PSBLAS library, and allows the use of advanced features, such as operator overloading and derived data type definition. A similar approach has already been advocated by a number of authors, e.g., Machiels and Deville [1997]. Moreover, the Fortran 90 facilities for dynamic memory management greatly enhance the usability of the PSBLAS subroutines. In this way, the library can take care of runtime memory requirements that are quite difficult or even impossible to predict at implementation or compilation time.

The presentation of this paper follows the general structure of the proposal for serial Sparse BLAS [Duff et al. 1997], which in turn is based on the proposal for BLAS on dense matrices [Lawson et al. 1979; Dongarra et al. 1988; 1990]. In particular, Section 2 gives an overview on the main components of the PSBLAS library. Section 3 describes the domain distribution strategy and the data structures for representing the local matrix storage and interprocess communication patterns. Section 4 presents the principal linear algebra operations of the PSBLAS library. Sections 5 and 6 analyze the main computational and auxiliary subroutines, respectively. Section 7 gives two examples of algorithms written through PSBLAS. Section 8 presents some experimental results. Section 9 contains a discussion of related work, and Section 10 contains remarks and outlines future work.

2. GENERAL OVERVIEW

We have designed the PSBLAS library to handle the implementation of iterative solvers for sparse linear systems on distributed-memory parallel computers. The system coefficient matrix A must be square; it may be real or complex, nonsymmetric, and its sparsity pattern needs not to be symmetric. The serial computation parts are based on the serial sparse BLAS, so that any extension made to the data structures of the serial kernels is available to the parallel version. The overall design and parallelization strategy have been influenced by the structure of the ScaLAPACK parallel library [Choi et al. 1995].

The layered structure of the PSBLAS library is shown in Figure 1. The work presented in this paper focuses on the Fortran 90 layer immediately below the application layer; we give two examples of iterative solvers built through the PSBLAS routines in Section 7. Lower layers of the library indicate an encapsulation relationship with upper layers. The serial parts of the computation on each process are executed through calls to the serial

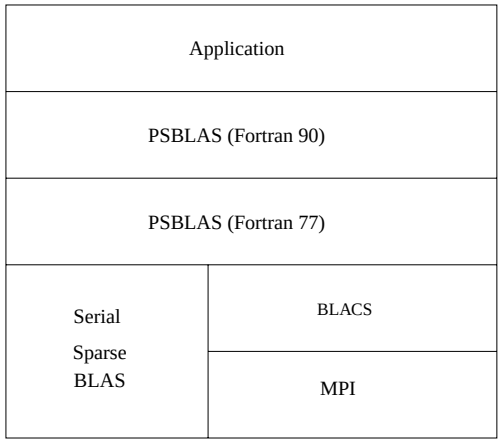

Fig. 1. PSBLAS library components hierarchy.

sparse BLAS subroutines; an implementation of the serial routines is included in our reference implementation. To enhance the usability of the code, we build the Fortran 90 interface on top of the Fortran 77 layer.

In a similar way, the interprocess message exchanges are implemented through the Basic Linear Algebra Communication Subroutines (BLACS) library [Dongarra and Whaley 1995] that guarantee a portable and efficient communication layer. The Message Passing Interface code is encapsulated within the BLACS layer. We did not find significant performance advantages with direct use of current implementations of MPI in the PSBLAS communication layer. However, we intend to reevaluate the BLACS vs. MPI alternative, as soon as efficient implementations of MPI-2 become available.

The PSBLAS library consists of two classes of subroutines, i.e., the *computational routines* and the *auxiliary routines*. The computational routine set includes:

—Sparse-matrix-by-dense-matrix product;

—Sparse triangular systems solution for block diagonal matrices;

—Vector and matrix norms;

—Dense matrix sums;

—Dot products.

The auxiliary routine set includes:

—Communication descriptors allocation;

—Dense and sparse matrix allocation;

—Dense and sparse matrix build and update;

—Sparse matrix and data distribution preprocessing.

The current implementation of PSBLAS addresses a distributed-memory execution model operating with message passing. However, the overall design does not preclude different implementation paradigms, such as those based on a shared-memory model.

3. MAIN PSBLAS DATA STRUCTURES

3.1 Library Design Choices

In any distributed-memory application, the data structures that represent the partition of the computational problem are essential to the viability and efficiency of the entire parallel implementation. The criteria guiding the decomposition choices are:

(1) maximizing load balancing,

(2) minimizing communication costs,

(3) optimizing the efficiency of the serial computation parts.

For *dense* linear algebra algorithms the ScaLAPACK library [Choi et al. 1995] has demonstrated that a block-cyclic distribution of the row and column index spaces is general and powerful enough to achieve a good compromise among the various factors that affect parallel computation performance.

The PSBLAS library addresses parallel *sparse* iterative solvers typically arising in the numerical solution of PDEs. In these instances, it is necessary to pay special attention to the structure of the problem from which the application originates. The nonzero pattern of a matrix arising from the discretization of a PDE is influenced by various factors, such as the shape of the domain, the discretization strategy, and the equation/unknown ordering. The matrix itself can be interpreted as the adjacency matrix of the graph associated with the discretization mesh; this characteristic will be discussed in Section 3.4.

The allocation of the coefficient matrix for the linear system is based on the "owner computes" rule: the associated variable of each mesh point is assigned to a process that will own the corresponding row in the coefficient matrix and will carry out all related computations. This allocation strategy is equivalent to a partition of the discretization mesh into *subdomains*. PSBLAS routines support any distribution that keeps together the coefficients of each matrix row; there are no other constraints on the variable assignment. The available distribution strategies include data distributions commonly used in ScaLAPACK such as `CYCLIC(N)` and `BLOCK`, as well as completely arbitrary assignments of equation indices to processes. Dense vectors conform to sparse matrices, i.e., the entries of a vector follow the same distribution of the matrix rows.

We assume that the sparse matrix is built in parallel, where each process generates its own portion. We never require that the entire matrix be available on a single node. However, it is possible to hold the entire matrix

in one process and distribute it explicitly,[1] even though the resulting bottleneck would make this option unattractive in most cases.

The storage scheme used for the computational parts pertaining to each process conforms to the storage formats defined in the serial sparse BLAS proposal [Duff et al. 1997]. The data structures that describe the local matrix storage are kept separated from those used for representing the communication pattern. This choice satisfies the encapsulation relations among the PSBLAS layers as described in Section 2.

3.2 Communication Descriptors

Once the distributed sparse matrix has been allocated and built, it is necessary to arrange the data structures that will be used to perform interprocess communications during the execution of routines such as matrix-vector products, preconditioners, and scalar products. The detailed contents of these data structures depend on the sparsity pattern of the coefficient matrix.

Our computational model implies that the data allocation on the parallel distributed-memory machine is guided by the structure of the physical model, and specifically by the discretization mesh of the PDE. Each point of the discretization mesh will have (at least) one associated equation/variable, and therefore one index. We say that point i *depends* on point j if the equation for a variable associated with i contains a term in j, or equivalently if $a_{ij} \neq 0$. After the partition of the discretization mesh into *subdomains* assigned to the parallel processes, we classify the points of a given subdomain as the following.

Internal. An internal point of a given domain *depends* only on points of the same domain. If all points of a domain are assigned to one process, then a computational step (e.g., a matrix-vector product) of the equations associated with the internal points requires no data items from other domains and no communications.

Boundary. A point of a given domain is a boundary point if it *depends* on points belonging to other domains.

Halo. A halo point for a given domain is a point belonging to another domain such that there is a boundary point which *depends* on it. Whenever performing a computational step, such as a matrix-vector product, the values associated with halo points are requested from other domains. A boundary point of a given domain is a halo point for (at least) another domain; therefore the cardinality of the boundary points set denotes the amount of data sent to other domains.

Overlap. An overlap point is a boundary point assigned to multiple domains. Any operation that involves an overlap point has to be replicated for each assignment. This may be acceptable, for example to

[1]In our prototype implementation we provide sample scatter/gather routines.

accelerate the convergence rate of the overall iterative method through an improvement of the preconditioning task [Filippone et al. 1992].

We denote the sets of internal, boundary and halo points for a given subdomain by $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{H}$. Each subdomain is assigned to one process; each process usually owns one subdomain, although the user may choose to assign more than one subdomain to a process. If each process i owns one subdomain, the number of rows in the local sparse matrix is $|\mathcal{I}_i| + |\mathcal{B}_i|$, and the number of local columns (i.e., those for which there exists at least one nonzero entry in the local rows) is $|\mathcal{I}_i| + |\mathcal{B}_i| + |\mathcal{H}_i|$.

The representation of the points of a subdomain assigned to a process is stored into integer arrays; the internal format is described in more detail in Cerioni et al. [1996]. Upon each invocation of a PSBLAS subroutine, we assume that we have a single active set of communication descriptors. This assumption is not too restrictive in view of the typical applications we address. Indeed, if the library is used in the context of a PDE solver, we often have multiple equations defined on the same discretization mesh, so that the coefficient matrices of the linear systems have the same sparsity pattern.

The definition of the communication descriptors is the following.

MATRIX_DATA A vector containing some general information, such as the descriptor status, the size of the global matrix, the number of local rows and columns, and the BLACS communication context.

HALO_INDEX The local list of the indices of halo points that have to be exchanged with other processes. For each process, this list contains the process identifier, the number and indices of points to be received, and the number and indices of points to be sent.

OVERLAP_INDEX A (local) list of the overlap points, organized in groups like the halo index descriptor; the format is identical to that of the halo descriptor.

OVERLAP_ELEM A (local) list containing for each overlap point its local index and the number of processes sharing it. This information is implicitly available in OVERLAP_INDEX; it is computed and stored separately at initialization time for efficiency reasons.

During the setup phase, we use two auxiliary arrays that keep track of the mapping between local and global indices. The Fortran 90 language allows a convenient packing of the necessary data structures inside a single *derived data type* as follows:

```
TYPE DESC_TYPE
  INTEGER, POINTER  :: MATRIX_DATA(:), HALO_INDEX(:),&
    & OVERLAP_ELEM(:), OVERLAP_INDEX(:),&
    & LOC_TO_GLOB(:), GLOB_TO_LOC(:)
END TYPE DESC_TYPE
```

3.3 Sparse Matrix Storage

The user defines matrix entries in terms of the global equation numbering. Each process in the parallel machine builds the sparse matrix rows that are assigned to it by the user. Once the build step is completed, the local part of the matrix undergoes a preprocessing operation. During this step the global numbering scheme is converted into the local numbering scheme, and the local sparse matrix representation is converted to a format suitable for subsequent computations. To carry out the internal storage conversion step and all other operations involving local sparse matrix computations, we rely on serial sparse BLAS routines [Duff et al. 1997; Carney et al. 1994]. Duff et al. [1997] present a detailed discussion about the rationale for the sparse matrix representation shown below. In particular, they describe the format of the permutation vectors PR and PC. These permutations arise in various sparse storage format conversions that may impose a renumbering of the local equations and variables to achieve the desired runtime efficiency. In PSBLAS, the sparse matrix storage conforms to the following Fortran 90 implementation of the serial sparse data structures:

```
TYPE D_SPMAT
   INTEGER M, K
   CHARACTER*5 FIDA
   CHARACTER*11 DESCRA
   INTEGER INFOA(10)
   REAL(KIND(1.D0)), POINTER  :: ASPK(:)
   INTEGER, POINTER :: IA1(:), IA2(:), PR(:), PC(:)
END TYPE D_SPMAT
```

Complex matrices have a similar structure, with the appropriate type declaration for member ASPK. Although the double complex data type is not strictly part of the Fortran 77 standard, it is available on almost all modern compilers.

3.4 Data Allocation Strategies and Graph Partitioning

In Section 3.2 we mentioned that the user can take any decision about the allocation of matrix rows and associated variables to processes; indeed, some experimental tests were based on random assignments. However, each allocation choice will result in a different runtime efficiency. In PSBLAS, we assume that a good runtime efficiency can be achieved through the choice of a data allocation that aims at minimizing the execution time of matrix-vector products. To motivate this assessment, it is necessary to consider the computational and communication costs of target applications of the PSBLAS library.

Modern iterative solvers are typically based on Krylov-subspace approximations [Kelley 1995; Greenbaum 1997]. In most instances, the cost of each iteration is constant across iterations. It is made up of matrix-vector products, preconditioning operations, scalar products, and dense vector sums. By analyzing these operations we find that:

—Dense vector sums do not involve any communication.

—Scalar products have a communication cost that is determined by the number of processes in the parallel machine.

—Many preconditioners, such as Jacobi, SOR, diagonal scaling, and basic ILU variants, either do not require interprocess communications or are associated with matrices having the same sparsity pattern as that of the coefficient matrix A.

The above observations imply that the only balance criterion for dense vector operations is that the vector entries should be evenly distributed among processes. We also consider incomplete factorization preconditioners that operate locally, because they usually give a good compromise between parallelization efficiency and preconditioning effectiveness in terms of reduction of the number of iterations. It would be also possible to implement a global factorization preconditioner by using our communication descriptors, provided that some constraints are imposed on the fill-in generation. However, this choice would require heavy modifications on the triangular matrix T storage format. Moreover, the performance of the triangular system solution would be much lower. In summary, the applicability and efficacy of a global factorization preconditioner do not seem general enough to warrant its implementation at this time of the project. As a consequence of previous observations, with little loss of generality, we can direct our optimization efforts toward the improvement of the parallel implementation of the matrix-vector product. The search for the optimal data distribution for this basic operation can be modeled as a weighted graph partitioning problem because the coefficient matrix can be interpreted as the adjacency matrix of the graph associated with the discretization mesh.

Each node of the graph models one row in the coefficient matrix; its weight equals the number of nonzero coefficients, as this is proportional to the computational cost of evaluating its contribution to the matrix-vector product. Each arc of the graph models a nonzero coefficient that needs to acquire the value of one variable to complete the computation associated with an equation. We assume that the cost of communicating a variable value is constant, so that all arcs have the same (unitary) weight.

The optimality criteria for the graph partitioning problem are:

(1) The total weight of the graph nodes should be evenly distributed.

(2) The weight of the edges crossing partition boundaries should be minimized.

If the variance degree is not too large, or equivalently if the graph structure is sufficiently regular, the node weight criterion induces a distribution of the graph nodes that balances performance of vector operations. Each vector operation adds a fixed number of floating-point operations per matrix row, and this is equivalent to adding a constant value to each node weight.

The general graph partition problem is $\mathcal{NP}$-complete [Garey and Johnson 1979]. Heuristics for obtaining reasonably good partitions in an effective way are the subject of a vast body of literature, and are out of the scope of the present paper. (See Karypis and Kumar [1995], Hendrickson and Leland [1995], and Pothen et al. [1990], and references therein.) PSBLAS can be interfaced with popular tools such as Metis [Karypis and Kumar 1995] with obvious benefits in performance. However, the use of a graph partition tool is not strictly required by PSBLAS. In fact, for most problems of interest to our library, a surface-to-volume effect takes place, so that even a BLOCK distribution may be satisfactory. Moreover, it is likely that the partition of the discretization mesh can be guided by physical considerations. Note that many graph partitioning algorithms work on undirected graphs, which would correspond to matrices with a symmetric pattern, but this is not required by PSBLAS. In these instances, it would be appropriate (even if simplistic) to apply the partition tool to the graph associated with the nonzero pattern of $A + A^T$, and to apply the resulting distribution to the original matrix.

For these reasons, we designed the PSBLAS library with the intention of letting the user control the entire specification of the data allocation. The only constraint is that the data allocation strategy satisfies the "owner computes" paradigm discussed in Section 3.1.

The reader familiar with *domain decomposition methods* may have noticed that the terminology for classifying the mesh points in Section 3.2 and the discussion about our data allocation include some terms and issues which are typical of that research area. Indeed, an alternative way to summarize our allocation choices is to say that we recommend a data distribution strategy which is consistent with a domain decomposition preconditioner applied to a global solver.

4. PSBLAS OPERATIONS

For the description of the operations in the PSBLAS library we use the following definitions and assumptions.

A	is a sparse matrix distributed by rows as described above;
T	is a sparse matrix with triangular blocks on the main diagonal and zeros elsewhere; it is distributed by rows conforming to A;
B, C, X, Y	are dense matrices distributed by rows conforming to A;
x, y	are dense vectors distributed by rows conforming to A;
α, β	are scalars;
P, P_R, P_C	are block diagonal permutation matrices;
D	is a diagonal matrix distributed by rows conforming to A.

The block diagonal form of T is due to the choice of a factorization preconditioner that works only on the part of the matrix local to each process. Hence, each block extends across the rows of the matrix assigned to a process. This preconditioner is usually called *block Jacobi* (sometimes *block ILU*) or *overlapping additive Schwartz* depending on the existence of overlap among subdomains.

The permutation matrices P, P_R, P_C are in block diagonal form too, because they depend on the storage format chosen for the part of A local to each process. Indeed, they do not appear explicitly in calling sequences because they are encapsulated into the local part of A, as discussed in Section 3.3.

The main operations in the PSBLAS library are:

—Matrix-matrix products

$$C \leftarrow \alpha P_R A P_C B + \beta C$$

$$C \leftarrow \alpha P_R A^T P_C B + \beta C$$

—Triangular system solutions

$$C \leftarrow \alpha P_R T^{-1} P_C B + \beta C$$

$$C \leftarrow \alpha D P_R T^{-1} P_C B + \beta C$$

$$C \leftarrow \alpha P_R T^{-1} P_C D B + \beta C$$

$$C \leftarrow \alpha P_R T^{-T} P_C B + \beta C$$

$$C \leftarrow \alpha D P_R T^{-T} P_C B + \beta C$$

$$C \leftarrow \alpha P_R T^{-T} P_C D B + \beta C$$

—Matrix sums

$$Y \leftarrow \alpha X + \beta Y$$

—Scalar products

$$x^T y \quad \text{or} \quad x^H y$$

—Dense vector 1, 2 and infinity norms

$$\|x\|$$

—Sparse matrix infinity norm

$$\|A\|_\infty$$

—Halo data communications

—Overlap data updates

Internal and *boundary* points are computed by the process that owns them. *Halo* points are requested from the owning processes through data communications. For the *overlap* points, we have to distinguish between the operations where all processes, given a consistent input, compute the same result (e.g., dense matrix sum) from the operations where each process computes its own result (e.g., triangular system solution). The latter operations may require an additional step to recover the consistency of dense vector entries across processes. When overlap regions are nonempty, the triangular system solution implements the preconditioner described in Filippone et al. [1992]:

$$K^{-1}r = P_a P^T \begin{pmatrix} K_1^{-1} & 0 & 0 & 0 \\ 0 & K_2^{-1} & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & K_n^{-1} \end{pmatrix} Pr$$

where K_i is the incomplete factorization local to subdomain i, P is an operator that "stretches" the vector r replicating the elements of the overlapped regions, and P_a is a diagonal scaling operator to obtain the average of vector entries corresponding to overlap points. The preconditioner can be made symmetric by applying the square root of P_a on both sides of K_i.

The default behavior of the subroutines is to keep all vector entries consistent across all subroutine calls, although this may be overridden for efficiency reasons. As an example, the application of an ILU preconditioner requires two successive calls to the triangular system solution routine: the former for the L part, and the latter for the U part of the preconditioner. As the preconditioning step is a local operation, we may choose to restore consistency on overlap points only after the second call to the triangular system solver routine or, equivalently, by letting K_i be the combined application of both lower and upper triangle of the factorization.

5. COMPUTATIONAL SUBROUTINES

In this section we outline the Fortran 90 interface of the PSBLAS computational subroutines. It is worth observing that these computational kernels are built as wrappers around a Fortran 77 layer, which is described in more detail in Cerioni et al. [1996]. We choose the Fortran 90 interface because the object-oriented features available in this language greatly improve readability and usability of the parallel applications written through PSBLAS. Two examples of algorithms will be given in Section 7.

All PSBLAS subroutines are prefixed by F90_PS. The names of the following computational subroutines are generic interfaces to real and complex versions.

All computational subroutines must be called by all processes. There are no explicit barrier constructs; however all computational subroutines employ global communications for error checking, thus providing synchronization. In all subroutines, DESC_A is the communication descriptor of type DESC_TYPE as described in Section 3.2; the input parameters are ordered in such a way that the parameters occurring after DESC_A are optional. Sparse matrices A and T must be declared of type D_SPMAT. Input scalars such as ALPHA and BETA must have the same value on all processes; dense matrices and vectors such as X and x must have the POINTER attribute.

F90_PSDOT Computes the dot product (x, y) of two vectors

DOT = F90_PSDOT(X, Y, DESC_A)

F90_PSAXPBY Computes

$$Y \leftarrow \alpha X + \beta Y$$

CALL F90_PSAXPBY(ALPHA, X, BETA, Y, DESC_A)

F90_PSAMAX Computes the maximum absolute value, or infinity norm of a vector

$$amax \leftarrow \max_i \{|X(i)|\} = \|X\|_\infty.$$

AMAX = F90_PSAMAX(X, DESC_A)

F90_PSASUM Computes the 1-norm of a vector

$$asum \leftarrow \sum_i |X(i)| = \|X\|_1.$$

ASUM = F90_PSASUM(X, DESC_A)

F90_PSNRM2 Computes the 2-norm of a vector

$$nrm2 \leftarrow \|X\|_2.$$

NRM2 = F90_PSNRM2(X, DESC_A)

F90_PSNRMI Computes the infinity-norm of a distributed sparse matrix

$$nrmi \leftarrow \|A\|_\infty.$$

NRMI = F90_PSNRMI(A, DESC_A)

F90_PSSPMM Computes the sparse-matrix-by-dense-matrix product

$$Y \leftarrow \alpha P_R A P_C X + \beta Y$$

$$Y \leftarrow \alpha P_R A^T P_C X + \beta Y$$

The permutation matrices P_R and P_C are local, as they handle all details of the local storage format. They do not appear explicitly in the following call format because they are encapsulated into A, according to the derived data type discussed in Section 3.3.

CALL F90_PSSPMM(ALPHA, A, X, BETA, Y, DESC_A, TRANS)

F90_PSSPSM Computes the triangular system solution:

$$Y \leftarrow \alpha P_R T^{-1} P_C X + \beta Y$$

$$Y \leftarrow \alpha D P_R T^{-1} P_C X + \beta Y$$

$$Y \leftarrow \alpha P_R T^{-1} P_C D X + \beta Y$$

$$Y \leftarrow \alpha P_R T^{-T} P_C X + \beta Y$$

$$Y \leftarrow \alpha D P_R T^{-T} P_C X + \beta Y$$

$$Y \leftarrow \alpha P_R T^{-T} P_C D X + \beta Y$$

The triangular sparse matrix T is block diagonal. This is equivalent to the application of local ILU or IC preconditioning.

CALL F90_PSSPSM(ALPHA, T, X, BETA, Y, DESC_A, TRANS, UNITD, IOPT, D)

The character UNITD parameter denotes how to use the diagonal D (stored as a vector), which can be assumed unitary (and thus ignored), or applied on the left or on the right. The $IOPT$ parameter denotes whether the consistency of the overlap points for vector X has to be enforced or not.

6. AUXILIARY SUBROUTINES

The PSBLAS library contains a set of tools that define the parallel data structure, and allocate and assemble the matrices involved in the computation. The goal is to encapsulate the low-level details of the internal storage of the communication descriptors and sparse matrices. The requirements we place on the user that wants to write applications in PSBLAS are the following.

(1) The allocation of the index space among the processes is defined by means of the subroutine PARTS described below.

(2) The coefficients of the sparse matrix are given in terms of global numbering.

The choice of a subroutine argument aims to guarantee maximal flexibility; the user can define an arbitrary data allocation scheme and experiment various partition strategies in a simple way. The PSBLAS package includes

some predefined subroutines for simple distributions such as BLOCK and CYCLIC. The coefficients of the linear system should be generated in a very simple format (currently, Compressed sparse rows or Coordinate) that is handled through the subroutine F90_PSSPINS. The library will manage other storage schemes through the preprocessing facilities of the serial sparse BLAS [Duff et al. 1997].

For any application, the first PSBLAS routine invoked must be F90_PSDSCALL that initializes the environment. We also assume that the application has already initialized the BLACS environment; each PSBLAS process will identify itself by means of its BLACS task index.

Auxiliary subroutines must be called by all processes, with the exception of the insertion routines F90_PSSPINS, F90_PSSPUPD, and F90_PSDSINS. These routines are called independently by each process to act on the local parts of the relevant sparse or dense matrices. The processes will synchronize upon the subsequent (and required) call to F90_PSPASB routine.

The auxiliary subroutines currently include:

F90_PSDSCALL Allocates and initializes communication descriptors data structures on the basis of the user information that is provided through the subroutine PARTS;

CALL F90_PSDSCALL(M, N, PARTS, DESC_A, ICONTXT)

M and N are the global matrix dimensions: we kept them separate, even though currently we only support square matrices. ICONTXT is the BLACS communication context returned by the BLACS environment initialization routine.

PARTS This is a user-provided subroutine. The index space allocation among the processes is obtained through the library call

CALL PARTS(G_IDX,PROCS,NPROCS)

where G_IDX is the input global index from the library routine, PROCS(*) is the output vector containing the indices of the BLACS task(s) owing the given index, and NPROCS is the number of valid entries in PROCS. If NPROCS $>$ 1, we have an overlap point.

F90_PSSPALL Allocates data structures for Global Sparse Matrix. The user may also provide an estimate of the number of nonzero elements that have to be allocated for the local part of the sparse matrix.

CALL F90_PSSPALL(A, DESC_A, NNZ)

F90_PSSPINS Inserts a sparse block into a sparse matrix. A local sparse matrix *BLCK* in CSR or coordinate format is inserted into the local part of the matrix *A*. It is not necessary to pass a complete matrix block to this routine. For example, the user that wants to implement a finite element application can denote the equations element by element without any ordering constraint. PSBLAS is able to keep track of contributions from different elements into the same equation (row of the sparse matrix). Multiple contributions to the same matrix entry can be treated

by summing their values, by keeping just one of the specified values and ignoring the others, or by raising an error condition, under the user control.[2]

CALL F90_PSSPINS(A, BLCK,DESC_A, IA, JA)

F90_PSSPASB Assembles Sparse Matrix. It processes the sparse matrix and the descriptor data structure to put them into the final format that is suitable for the computational routines. This routine checks for errors that may have occurred during the insertion phase.

CALL F90_PSSPASB(A, DESC_A)

The descriptor is fully updated (on the basis of the nonzero structure of the global sparse matrix) through the call to this subroutine.

F90_PSSPFREE Deallocates the memory associated with the components of a sparse matrix.

CALL F90_PSSPFREE(A, DESC_A)

F90_PSDSCFREE Deallocates the memory associated with the components of a descriptor data structure.

CALL F90_PSDSCFREE(DESC_A)

F90_PSSPUPD Allows the *reuse* of an existing sparsity pattern. Its interface is identical to that of `F90_PSSPINS`. The use of `F90_PSSPUPD` is convenient in those instances in which the parallel application needs to solve a sequence of linear systems that have the same sparsity pattern and differ for the values of the coefficients.

Similar routines are provided for building dense matrices that are distributed conforming to the sparse matrix:

F90_PSDSALL Allocates a dense matrix; dense matrices must be declared with the `POINTER` attribute.

F90_PSDSINS Inserts a dense block into a dense matrix.

F90_PSDSASB Analogous to `F90_PSSPASB` for a dense matrix.

F90_PSDSFREE Deallocates a dense matrix.

The general structure of an application using PSBLAS is:

(1) Initialize the communication descriptors with `F90_PSDSCALL`; initialize the sparse matrix with `F90_PSSPALL`.

(2) Loop on all mesh points owned by the current process; build their equations; and insert them into the sparse matrix and right-hand side through calls to `F90_PSSPINS` and `F90_PSDSINS`.

[2]This is done through the serial preprocessing routines; since this behavior is not strictly specified in the serial sparse BLAS, it can be viewed as an extension to that specification.

(3) Assemble the matrix and finalize the communication descriptors with F90_PSSPASB;

(4) Compute the preconditioner and call the iterative method.

If the same discretization mesh is reused, it is possible to repeat steps 2 and 3 by using F90_PSSPUPD. The order in which the mesh points are visited in step 2 is arbitrary. This is useful for finite element applications, where it may be desirable to loop over the elements, rather than over the equations. As an example, consider the sample application shown in Section 5 of Machiels and Deville [1997]: the Poisson matrix assemble loop in function assemble_matrix_a runs through the elements. By recasting the sample code into PSBLAS calls, we obtain:

```
DO iel=1,ael%nel
  blck = give_matrix_e(ael%el(iel),which)
  ! assume IMIN is the lowest row index among ael%index(i,iel)
  CALL F90_PSSPINS(a,imin,1,blck,desc_a)
ENDDO
CALL F90_PSSPASB(a,desc_a)
```

This example assumes that the original function give_matrix_e has been rewritten to return a sparse matrix in coordinate format instead of a dense matrix as in Machiels and Deville [1997]. Actual PSBLAS code may look slightly different if optional parameters are used.

If the data distribution is independent of the discretization mesh structure, for example when using a BLOCK distribution, the above application structure has no serial bottlenecks. If a graph partitioning package is used, the cost of the setup phase depends on the graph partitioning routine. If the subroutine is serial, there is a serial bottleneck both in terms of processing time and memory space, because one of the processes will hold the structure of the entire discretization mesh. For many applications this would not be a serious drawback, because the linear solver itself is a single step in an outer solution algorithm, and often many (if not all) consecutive steps share the same mesh topology.

The serial graph partitioning approach is not very suitable to applications that use adaptive meshes with fast rates of change; extensions to PSBLAS for such applications are currently being investigated [Filippone et al. 1999].

7. ITERATIVE METHODS

To illustrate the use of the library routines, we describe in Tables I and II the templates for the CG and Bi-CGSTAB methods from Barrett et al. [1994], with local ILU preconditioning and normwise backward error stopping criterion [Arioli et al. 1992]. The examples show the high readability and usability features of the PSBLAS with Fortran 90 interface. The mathematical formulation of the algorithms is quite comparable to the PSBLAS implementation.

Table I. Sample CG Implementation

Template [Barrett et al. 1994]	PSBLAS Implementation

Compute $r^{(0)} = b - Ax^{(0)}$	`call f90_psaxpby(one,b,zero,r,desc_a)`
	`call f90_psspmm(-one,A,x,one,r,desc_a)`
	`bni = f90_psamax(b,desc_a)`
	`ani = f90_psnrmi(A,desc_a)`
	`rho = zero`

for $i = 1, 2, \ldots$	`do it = 1, itmax`
solve $Mz^{(i-1)} = r^{(i-1)}$	`call f90_psspsm(one,L,r,zero,w,desc_a)` `call f90_psspsm(one,U,w,zero,z,desc_a)`
$\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$	`rho_old = rho` `rho = f90_psdot(r,z,desc_a)`
if $i = 1$	`if (it == 1) then`
$p^{(1)} = z^{(0)}$	`call f90_psaxpby(one,z,zero,p,desc_a)`
else	`else`
$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$	`beta = rho/rho_old`
$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$	`call f90_psaxpby(one,z,beta,p,desc_a)`
endif	`endif`
$q^{(i)} = Ap^{(i)}$	`call f90_psspmm(one,A,p,zero,q,desc_a)`
$\alpha_i = \rho_{i-1} / p^{(i)^T} q^{(i)}$	`sigma = f90_psdot(p,q,desc_a)` `alpha = rho/sigma`
$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$	`call f90_psaxpby(alpha,p,one,x,desc_a)`
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$	`call f90_psaxpby(-alpha,q,one,r,desc_a)`

Check convergence:
$\|r^{(i)}\|_\infty \leq \epsilon(\|A\|_\infty \cdot \|x^{(i)}\|_\infty + \|b\|_\infty)$
	`rni = f90_psamax(r,desc_a)`
	`xni = f90_psamax(x,desc_a)`
	`err = rni/(ani*xni+bni)`
	`if (err.le.eps) return`

end	`enddo`

Efficiency improvements can be obtained from this basic implementation through optional parameters that are available in the subroutine interfaces.

8. EXPERIMENTAL RESULTS

The current implementation of PSBLAS mostly strives to achieve correctness and usability. Many performance improvements are still possible, especially in the local computation parts that are handled by the serial Sparse BLAS code. Nonetheless, the efficiency of the library is already satisfactory. As examples, in this paper we show the performance of the Conjugate Gradient and Bi-CGSTAB methods obtained on an IBM SP2 equipped with 160MHz Power2 nodes and SPSwitch interconnection network. All tests have been carried out using a local ILU preconditioner with no fill-in. For the runs with $NP = 1$ we have used the same code as that for the experiments with multiple processes; however the implementation of the routines is optimized, so that the solver code is equivalent to a scalar code implemented through the serial sparse BLAS.

Table II. Sample Bi-CGSTAB Implementation

Template [Barrett et al. 1994]	PSBLAS Implementation
Compute $r^{(0)} = b - Ax^{(0)}$	`call f90_psaxpby(one,b,zero,r,desc_a)` `call f90_psspmm(-one, A, x, one,r,desc_a)`
Choose q (e.g., $q = r^{(0)}$)	`call f90_psaxpby(one,r,zero,q,desc_a)`
	`bni = f90_psamax(b,desc_a)` `ani = f90_psnrmi(A,desc_a)`
for $i = 1, 2, \ldots$	`do it = 1, itmax`
$\rho_{i-1} = q^T r^{(i-1)}$	`rho_old = rho` `rho = f90_psdot(q,r,desc_a)`
if $i = 1$	`if (it == 1) then`
$p^{(1)} = r^{(0)}$	`call f90_psaxpby(one,r,zero,p,desc_a)`
else	`else`
if $\rho = 0$ **failure**	`if (rho==0) stop`
$\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$	`beta = (rho/rho_old)(alpha/omega)`
$p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1}v^{(i-1)})$	`call f90_psaxpby(-omega,v,one,p,desc_a)` `call f90_psaxpby(one,r,beta,p,desc_a)`
endif	`endif`
solve $M\hat{p} = p^{(i-1)}$	`call f90_psspsm(one,L,p,zero,w,desc_a)` `call f90_psspsm(one,U,w,zero,phat,desc_a)`
$v^{(i)} = A\hat{p}$	`call f90_psspmm(one,A,phat,zero,v,desc_a)`
$\alpha_i = \rho_{i-1}/q^T v^{(i)}$	`alpha = f90_psdot(q,v,desc_a)` `alpha = rho/alpha`
$s = r^{(i-1)} - \alpha v^{(i)}$	`call f90_psaxpby(one,r,zero,s,desc_a)` `call f90_psaxpby(-alpha,v,one,s,desc_a)`
solve $M\hat{s} = s$	`call f90_psspsm(one,L,s,zero,w,desc_a)` `call f90_psspsm(one,U,w,zero,shat,desc_a)`
$t = A\hat{s}$	`call f90_psspmm(one,A,shat,zero,t,desc_a)`

$\omega_i = t^T s / t^T t$	`omega = f90_psdot(t,s,desc_a)`
	`temp = f90_psdot(t,t,desc_a)`
	`omega=omega/temp`

$x^{(i)} = x^{(i-1)} + \alpha_i \hat{p} + \omega_i \hat{s}$	`call f90_psaxpby(alpha,phat,one,x,desc_a)` `call f90_psaxpby(omega,shat,one,x,desc_a)`
$r^{(i)} = s - \omega_i t$	`call f90_psaxpby(one,s,zero,r,desc_a)` `call f90_psaxpby(-omega,t,one,r,desc_a)`

Check convergence:
$\|r^{(i)}\|_\infty \leq \epsilon(\|A\|_\infty \cdot \|x^{(i)}\|_\infty + \|b\|_\infty)$
	`rni = f90_psamax(r,desc_a)`
	`xni = f90_psamax(x,desc_a)`
	`err = rni/(ani*xni+bni)`
	`if (err.le.eps) return`

end	`enddo`

For the Conjugate Gradient test, we have generated a standard Poisson matrix on a cubic domain that has been simply partitioned by assigning blocks of rows of equal sizes to the various processes. We report in Table III the time per iteration for various cube edge dimensions L as a function of the number of processes NP. This table shows that the speedup is acceptable, provided that the number of mesh points (cube size) is large enough to offset the system overhead and the communication costs. The number of

Table III. CG: Constant Problem Size, Cube of Size L

	Time per Iteration			
NP	$L = 30$	$L = 40$	$L = 50$	$L = 80$
1	0.0549	0.1165	0.2261	0.9304
2	0.0369	0.0674	0.1246	0.4841
4	0.0334	0.0463	0.0769	0.2826
8	0.0388	0.0364	0.0528	0.1659
16	0.0373	0.0333	0.0441	0.0990

iterations to achieve convergence changes over the various test cases, but typically by no more than 10%.

For the Bi-CGSTAB tests we used a three-dimensional problem from the seven-point difference discretization of an operator similar to an example discussed in Kelley [1995]:

$$(Lu) = -(u_{xx} + u_{yy} + u_{zz}) + a_1 u_x + a_2 u_y + a_3 u_z + bu = f$$

We executed two series of tests. In the first series we chose a simple BLOCK distribution of the rows of the matrix, and varied the mesh size and the number of processors. Table IV shows the time to build the matrix vs. the time to solve the linear system. All time values are in seconds. For each column the discretization mesh size is kept constant and the number of processors is varied (*constant problem size*). The results demonstrate something expected, i.e., the efficiency of the parallelization is acceptable only when the problem size is large enough to keep the workload sufficiently high on all processors. As discussed in Section 6, the build time is affected by the size of the matrix which is used as a parameter of the `F90_PSSPINS` function. In particular, the build times shown in Table IV refer to the simple strategy of building the matrix one row at a time; the build time can be improved by applying the insertion routines to blocks comprising a few rows at a time.

In the second series of tests we partitioned the discretization mesh into subdomains, and assigned these subdomains to processors. Table V shows the average time per iteration and the number of iterations. For each column, the discretization mesh is scaled so as to keep constant the number of mesh points per processor, i.e., we refine the discretization as we have more processors available. This table shows that the time per iteration depends mostly on the number of mesh points per processor, provided that this number is sufficiently large.

9. RELATED WORK

The attempt to standardize serial sparse computations is one of the tasks in the current activity of the BLAS Technical Forum.[3] Furthermore, much research work is devoted to the implementation of iterative solvers for

[3] `http://www.netlib.org/blast`

Table IV. Bi-CGSTAB: Fixed Problem Size, BLOCK Distribution

	Build Time / Solve Time			
	Discretization Mesh			
NP	$20 \times 20 \times 20$	$30 \times 30 \times 30$	$40 \times 40 \times 40$	$80 \times 80 \times 80$
1	0.63 / 0.45	2.25 / 2.04	5.58 / 6.32	48.1 / 80.1
2	0.37 / 0.44	1.26 / 1.41	3.08 / 3.54	27.2 / 41.7
4	0.36 / 0.72	0.70 / 1.10	1.66 / 2.50	14.2 / 23.3
8	0.48 / 1.07	0.40 / 1.15	0.90 / 2.04	7.4 / 20.7

Table V. Bi-CGSTAB: Scaled Problem Size with N Mesh Points per Processor

	Time per Iteration / Number of Iterations			
NP	$N = 8K$	$N = 27K$	$N = 64K$	$N = 125K$
1	0.0307 / 17	0.0870 / 23	0.2104 / 30	0.4124 / 32
2	0.0743 / 24	0.1051 / 31	0.2321 / 39	0.4455 / 48
4	0.0608 / 24	0.1170 / 44	0.2515 / 40	0.4663 / 47
8	0.0617 / 32	0.1312 / 58	0.2707 / 52	0.4884 / 47

sparse linear systems on distributed-memory machines. One of the most complete works is the PETSc programming environment [Balay et al. 1995] which is based mostly on the C language. The PETSc approach has many interesting features, such as the provision for setup and support routines, and ease of implementation of new iterative methods. We feel PSBLAS is somewhat complementary to PETSc, in that it addresses more closely the needs of the Fortran user community.

Aztec [Hutchinson et al. 1998] is another software package developed at Sandia National Laboratories that simplifies the parallelization of sparse linear systems solution. Aztec's approach to domain partitioning—conversion from global to local data and keeping track of partitioning information—is essentially identical to that of PSBLAS. The most important difference is that Aztec is designed primarily for C and Fortran 77 users, and does not provide object-oriented Fortran 90 capabilities.

Unlike PETSc and Aztec approaches, object-oriented design methodologies are at the basis of the Diffpack project [Bruaset and Langtangen 1997]. Its authors propose the use of C++ instead of C and demonstrate that object-oriented numerical software is easy to use, flexible, maintainable, and can be computationally efficient even in the field of sparse matrix solvers.

The design of sparse iterative solvers for parallel machines is also a key activity in the PARASOL and PINEAPL projects funded by the European Union under the IV Framework ESPRIT program. The PARASOL project,[4] coordinated by Pallas GmbH (D), is quite different from PSBLAS in that it takes a long-term and ambitious view. In particular, PARASOL aims at

[4]http://www.pallas.de/outpage/out-par.html

providing a new and comprehensive framework, spanning from the finite element model level to the linear solver level. The PSBLAS effort has a more focused approach on the implementation of the parallel linear solvers. This choice allows us to achieve some interesting results even if more restricted than the PARASOL purposes.

The PINEAPL project,[5] coordinated by NAG Ltd. (UK), aims at building a library of general-purpose software for parallel distributed-memory machines. In the plans of the designers, the PINEAPL library should take into account the requirements of a number of end-user applications, including fluid dynamics simulations, electromagnetic scattering, and geometrical shape optimization. The PINEAPL library will eventually be integrated into the NAG Parallel Library. Even though the scope of the PINEAPL project is broader than that of PSBLAS library, the linear algebra part bears many similarities with our work in terms of data distribution choices and types of operations implemented. However, the PINEAPL linear algebra solvers are written in pure Fortran 77, so they cannot exploit Fortran 90 data hiding and operator overloading as PSBLAS does. Moreover, the PSBLAS support routines for building the sparse matrix give much more flexibility and freedom to the user.

Finally, it should be observed that the PSBLAS project has strongly influenced the design of the sparse iterative solvers in the IBM Parallel Engineering and Scientific Subroutine Library product.[6]

10. CONCLUSIONS

This paper presents a software framework for enabling easy, efficient, and portable parallel implementations of sparse matrix computations. The main targets of the PSBLAS library are applications that require the solution of sparse linear systems via preconditioned iterative methods. The main features of this framework are the use of various programming paradigms for the different implementation layers (of particular evidence, it is the use of advanced features of Fortran 90, such as operator overloading and derived data type definition), and the flexibility about computational domain distribution among the parallel processors. The library is built on top of the serial sparse BLAS, and in principle it could be interfaced with a variety of partitioning algorithms. Although we have designed the interface having in view an SPMD programming model on distributed-memory machines, we do not exclude different implementations, such as those based on a shared-memory model of computations. Indeed, one of the main design and implementation goals was to provide a flexible framework and library that could easily sustain further developments.

We are completing the integration of the PSBLAS library with supports for parallel applications that require dynamic mesh refinements [Filippone

[5]http://www.nag.co.uk/projects/PINEAPL.html
[6]http://www.rs6000.ibm.com/software/

et al. 1999]. Other extensions could improve the current version of the PSBLAS library without major changes in the code. Improvements to the preprocessing routines should allow the user to specify some information about the characteristics and exploitation of the sparse matrix (e.g., symmetric storage, block entries, use of the transpose matrix), or other properties of the application (e.g., approximately how many iterations will be performed, how many times the solver will be called with the same sparsity pattern). Another area of future work is the implementation of new preconditioners, ranging from basic variations of those available today to completely new ones.

A prototype version of the library software is available from `www.ce.uniroma2.it/psblas`.

ACKNOWLEDGMENTS

We wish to thank Salvatore Tucci of the Università di Roma Tor Vergata for his help and support in this cooperative work, and Fabio Cerioni, Stefano Maiolatesi, and Dario Pascucci for their contributions in the implementation of various parts of the PSBLAS library. We are grateful to Iain Duff and Carlo Vittoli for the discussions held in the design phase of the PSBLAS library. The insightful comments of the editor John Reid and the anonymous referees were very helpful in improving the quality of this paper and enlightening some directions for future work.

REFERENCES

ARIOLI, M., DUFF, I., AND RUIZ, D. 1992. Stopping criteria for iterative solvers. *SIAM J. Matrix Anal. Appl. 13*, 1 (Jan.), 138–144.

BALAY, S., GROPP, W., MCINNES, L. C., AND SMITH, B. 1995. PETSc 2.0 user manual. ANL-95/11 - Revision 2.0.22. Argonne National Laboratory, Argonne, IL.

BARRETT, R., BERRY, M., CHAN, T., DEMMEL, J., DONAT, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND VORST, H. V. D. 1994. *Templates for the Solution of Linear Systems.* SIAM, Philadelphia, PA.

BRUASET, A. M. AND LANGTANGEN, H. P. 1997. Object-oriented design of preconditioned iterative methods in diffpack. *ACM Trans. Math. Softw. 23*, 1, 50–80.

CARNEY, S., HEROUX, M. A., LI, G., AND WU, K. 1994. A revised proposal for a sparse BLAS toolkit. Tech. Rep. 94-034. Army High Performance Computing Research Center, Minneapolis, MN.

CERIONI, F., COLAJANNI, M., FILIPPONE, S., AND MAIOLATESI, S. 1996. PSBLAS user's guide. RI.96.11 (April). Univ. of Rome—Tor Vergata, Rome, Italy.

CHOI, J., DEMMEL, J., DHILLON, I., DONGARRA, J. J., OSTROUCHOV, S., PETITET, A., STANLEY, K., WALKER, D. W., AND WHALEY, R. C. 1995. ScaLAPACK: A portable linear algebra library for distributed memory computers—design issues and performance. In *Proceedings of the second international workshop on Applied Parallel Computing* (PARA '95, Lyngby, Denmark), J. J. Dongarra, K. Masden, and J. Waśniewski, Eds. Springer-Verlag, New York, NY, 95–106.

DONGARRA, J. J. AND WHALEY, R. C. 1995. A user's guide to the BLACS v1.0. LAPACK Working Note 94 Technical Report CS-95-281. Department of Computer Science, University of Tennessee, Knoxville, TN. http://www.netlib.org/lapack/lawns

DONGARRA, J. J., DU CROZ, J. J., HAMMARLING, S., AND DUFF, I. S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1 (Mar.), 1–17.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. J. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw. 14*, 1 (Mar.), 1–17.

DUFF, I. S., MARRONE, M., RADICATI, G., AND VITTOLI, C. 1997. Level 3 basic linear algebra subprograms for sparse matrices: A user-level interface. *ACM Trans. Math. Softw. 23*, 3, 379–401.

FILIPPONE, S., COLAJANNI, M., AND PASCUCCI, D. 1999. An object-oriented environment for sparse parallel computation on adaptive grids. In *Proceedings of IPPS/SPDP* (San Juan, Puerto Rico, Apr.).

FILIPPONE, S., MARRONE, M., AND RADICATI DI BROZOLO, G. 1992. Parallel preconditioned conjugate-gradient type algorithms for general sparsity structures. *Inter. J. Comput. Math. 40*, 159–167.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY.

GREENBAUM, A. 1997. *Iterative Methods for Solving Linear Systems*. SIAM Frontiers in Applied Mathematics Series. SIAM, Philadelphia, PA.

HENDRICKSON, B. AND LELAND, R. 1995. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput. 16*, 2 (Mar.), 452–469.

HUTCHINSON, S., PREVOST, L., SHADID, J., TONG, C., AND TUMINARO, R. 1998. Aztec user's guide: Version 2.0. Sandia National Laboratories, Livermore, CA.

KARYPIS, G. AND KUMAR, V. 1995. METIS: Unstructured graph partitioning and sparse matrix ordering system. Computer Science Department, Univ. of Minnesota, Minneapolis, MN. http://www.cs.umn.edu/ karypis

KELLEY, C. T. 1995. *Iterative Methods for Linear and Nonlinear Equations.* SIAM Frontiers in Applied Mathematics Series. SIAM, Philadelphia, PA.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. 5*, 3, 308–323.

MACHIELS, L. AND DEVILLE, M. O. 1997. Fortran 90: An entry to object-oriented programming for the solution of partial differential equations. *ACM Trans. Math. Softw. 23*, 1, 32–49.

METCALF, M. AND REID, J. 1990. *Fortran 90 Explained*. Oxford University Press, Inc., New York, NY.

POTHEN, A., SIMON, H. D., AND LIOU, K.-P. 1990. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl. 11*, 3 (July), 430–452.

Received: April 1998; revised: April 1999, April 2000, June 2000, July 2000; accepted: July 2000