An Object-Oriented Framework for Block Preconditioning

Edmond Chow
University of Minnesota
and
Michael A. Heroux
Silicon Graphics, Inc.

General software for preconditioning the iterative solution of linear systems is greatly lagging behind the literature. This is partly because specific problems need specific matrix and preconditioner data structures in order to be solved efficiently; i.e., multiple implementations of a preconditioner with specialized data structures are required. This article presents a framework to support preconditioning with various, possibly user-defined, data structures for matrices that are partitioned into blocks. The main idea is to define data structures for the blocks, and an upper layer of software which uses these blocks transparently of their data structure. This transparency can be accomplished by using an object-oriented language. Thus various preconditioners, such as block relaxations and block incomplete factorizations, only need to be defined once, and will work with any block type. In addition, it is possible to transparently interchange various approximate or exact techniques for inverting pivot blocks, or solving systems whose coefficient matrices are diagonal blocks. This leads to a rich variety of preconditioners that can be selected. Operations with the blocks are performed with optimized libraries or fundamental data types. Comparisons with an optimized Fortran 77 code on both workstations and Cray supercomputers show that this framework can approach the efficiency of Fortran 77, as long as suitable block sizes and block types are chosen.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear systems (direct and iterative methods), Matrix inversion, Sparse and very large systems*; G.4 [**Mathematics of Computing**]: Mathematical Software; D.1.5 [**Software**]: Programming languages—*Object-oriented programming*

General Terms: Design

This work was supported in part by the National Science Foundation under grant NSF/CCR-9214116 and in part by Silicon Graphics, Inc. and the Minnesota Supercomputer Institute. The software is available at `http://www.cs.umn.edu/~chow/bpkit.html`.
Name: Edmond Chow
Address: Department of Computer Science and Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN 55455, `chow@cs.umn.edu`
Name: Michael A. Heroux
Address: Mathematical Algorithms and Scalable Computing Group, Cray Research, Silicon Graphics, Inc., Eagan, MN 55121, `mamh@cray.com`

Additional Key Words and Phrases: Block matrices, preconditioners

1. INTRODUCTION

In the iterative solution of the linear system

$$Ax = b,$$

a *preconditioner* M is often used to transform the system into one which has better convergence properties, for example, in the left-preconditioned case,

$$M^{-1}Ax = M^{-1}b.$$

M^{-1} is referred to as the *preconditioning operator* for the matrix A and, in general, is a sequence of operations that somehow approximates the effect of A^{-1} on a vector.

Unfortunately, general software for preconditioning is seriously lagging behind methods being published in the literature. Part of the reason is that many methods do not have general applicability: they are not robust on general problems, or they are specialized and need specific information (e.g., general direction of flow in a fluids simulation) that cannot be provided in a general setting.

Another reason, one that we will deal with in this article, is that specific linear systems need specific matrix and preconditioner data structures in order to be solved efficiently; i.e., there need to be multiple implementations of a preconditioner with specialized data structures. For example, in some finite element applications, diagonal blocks have a particular but fixed sparse structure. A block SSOR preconditioner that needs to invert these diagonal blocks should use an algorithm suited to this structure. A block SSOR code that treats these diagonal blocks in a general way is not ideal for this problem.

When we encounter linear systems from different applications, we need to determine suitable preconditioning strategies for their iterative solution. Rather than code preconditioners individually to take advantage of the structure in each application, it is better to have a framework for software reuse. Also, a wide range of preconditionings should be available so that we can choose a method that matches the difficulty of the problem and the computer resources available.

This article presents a framework to support preconditioning with various, possibly user-defined, data structures for matrices that are partitioned into blocks. The main idea is to define data structures (called *block types*) for the blocks, and an upper layer of software which uses these blocks transparently of their data structure. Thus various preconditioners, such as block relaxations and block incomplete factorizations, only need to be defined once, and will work with any block type. These preconditioners are called *global preconditioners* for reasons that will soon become apparent. The code for these preconditioners is almost as readable as the code for their pointwise counterparts. New global preconditioners can be added in the same fashion.

Global preconditioners need methods (called *local preconditioners*) to approximately or exactly invert pivot blocks, or solve systems whose coefficient matrices are diagonal blocks. For example, a block stored in a sparse format might be inverted exactly, or an approximate inverse might be computed. Our design permits a variety of these inversion or solution techniques to be defined for each block type.

The transparency of the block types and local preconditioners can be implemented through polymorphism in an object-oriented language. Our framework, called BPKIT, currently implements block incomplete factorization and block relaxation global preconditioners, a dense and a sparse block type, and a variety of local preconditioners for both block types. Users of BPKIT will either use the block types that are available, or add block types and local preconditioners that are appropriate for their applications. Users may also define new global preconditioners that take advantage of the existing block types and local preconditioners. Thus BPKIT is not intended to be complete library software; rather it is a framework under which software can be specialized from relatively generic components.

It is appropriate to make some comments about why we use *block* preconditioning. Many linear systems from engineering applications arise from the discretization of coupled partial differential equations. The blocking in these systems may be imposed by ordering together the equations and unknowns at a single grid point, or those of a subdomain. In the first case, the blocks are usually dense; in the latter case, they are usually sparse. Experimental tests suggest it is very advantageous for preconditionings to exploit this block structure in a matrix [Chow and Saad 1997; Fan et al. 1996; Jones and Plassmann 1995; Kolotilina et al. 1991]. The relative robustness of block preconditioning comes partly from being able to solve accurately for the strong coupling within these blocks. From a computational point of view, these block matrix techniques can be more efficient on cached and hierarchical memory architectures because of better data locality. In the dense block case, block matrix data structures also require less storage. Block data structures are also amenable to graph-based reorderings and block scalings.

When approximations are also used for the diagonal or pivot blocks (i.e., approximations with local preconditioners are used), these techniques are specifically called *two-level* preconditioners [Kolotilina and Yeremin 1986], and offer a middle-ground between accuracy and simpler computations. Beginning with [Underwood] in 1976 and then [Axelsson et al. 1984] and [Concus et al. 1985] more than a decade ago, these preconditioners have been motivated and analyzed in the case of block tridiagonal incomplete factorizations combined with several types of approximate inverses, and have recently reached a certain maturity. Most implementations of these methods, however, are not flexible: they are often coded for a particular block size and inversion technique, and further, they are almost always coded for dense blocks.

The software framework presented here derives its flexibility from the use of an object-oriented language. We chose to use C++ [Stroustrup 1991] in real, 64-bit arithmetic. Other object-oriented languages are also appropriate. The framework is computationally efficient, since all operations involving blocks are performed with code that employs fundamental types, or with optimized Fortran 77 libraries such as the Level 3 BLAS [Dongarra et al. 1990], LAPACK [Demmel 1989], and the sparse BLAS toolkit [Carney et al. 1994]. By the same token, users implementing block types and local preconditioners may do so in practically any language, as long as the language can be linked with C++ by their compilers. BPKIT also has an interface for Fortran 77 users.

BPKIT is available at `http://www.cs.umn.edu/~chow/bpkit.html`. Other C++ efforts in the numerical solution of linear equations include LAPACK++ [Dongarra et al. 1993] for dense systems, and Diffpack [Bruaset and Langtangen 1997], ISIS++ [Clay 1997], SparseLib++ and IML++ [Dongarra et al. 1994] for sparse systems. It is also possible to use an object-oriented style in other languages [Eijkhout 1996; Machiels and Deville 1997; Smith et al. 1995].

In Section 2, we discuss various issues that arise when designing interfaces for block preconditioning and for preconditioned iterative methods in general. We describe the specification of the block matrix, the global and local preconditioners, the interface with iterative methods, and the Fortran 77 interface. In Section 3, we describe the internal design of BPKIT, including the polymorphic operations on blocks that are needed by global preconditioners. In Section 4, we present the results of some numerical tests, including a comparison with an optimized Fortran 77 code. Section 5 contains concluding remarks.

2. INTERFACES FOR BLOCK PRECONDITIONING

We have attempted to be general when defining interfaces (to allow for extensions of functionality), and we have attempted to accept precedents where we overlap with related software (particularly in the interface with iterative methods). For concreteness, we describe several methods which will be used in the numerical tests. Section 2 brings to light various issues in the software design of preconditioned iterative methods.

2.1 Block matrices

A matrix that is partitioned into blocks is called a *block matrix*. Although with BPKIT any storage scheme may be used to store the blocks that are not zero, the locations of these blocks within the block matrix must still be defined. The block matrix class (data type) that is available in BPKIT, called `BlockMat`, contains a pointer to each block in the block matrix. The pointers for each row of blocks (block row) are stored contiguously, with additional pointers to the first pointer for each block row. This is the analogy to the compressed sparse row data structure [Saad 1990] for pointwise matrices; pointers point to blocks instead of scalar entries. The global preconditioners in BPKIT assume that the `BlockMat` class is being used. It is possible for users to design new block matrix classes and to code new global preconditioners for their problems, and still use the block types and local preconditioners in BPKIT.

For the block matrix data structure described above, BPKIT provides conversion routines to that data structure from the Harwell-Boeing format [Duff et al. 1989]. There is one conversion routine for each block type (e.g., one routine will convert a Harwell-Boeing matrix into a block matrix whose blocks are dense). However, these routines are provided for illustration purposes only. In practice, a user's matrix that is already in block form (i.e., the nonzero entries in each block are stored contiguously) can usually be easily converted by the user directly into the `BlockMat` form.

To be general, the conversion routines allow two levels of blocking. In many problems, particularly linear systems arising from the discretization of coupled partial differential equations, the blockings may be imposed by ordering together the equations and unknowns at a single grid point and those of a subdomain. The latter blocking produces *coarse*-grain blocks, and the smaller, nested blocks are called *fine*-grain blocks. Figure 1 shows a block matrix of dimension 24 with coarse blocks of dimension 6 and fine blocks of dimension 2.

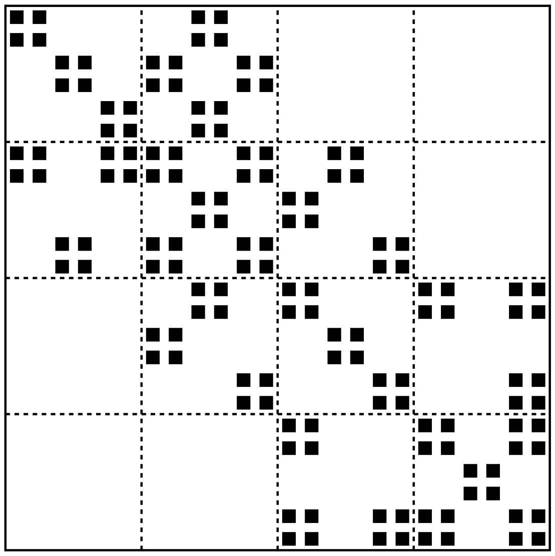
Fig. 1. Block matrix with coarse and fine blocks.

The blocks in BPKIT are the coarse blocks. Information about the fine blocks should also be provided to the conversion routines because it may be desirable to store blocks such that the coarse blocks themselves have block structure. For example, the variable block row (VBR) [Saad 1990] storage scheme can store coarse blocks with dense fine blocks in reduced space. Optimized matrix-vector product and triangular solve kernels for the VBR and other block data structures are provided in the sparse BLAS toolkit [Carney et al. 1994; Remington and Pozo 1996]. No local preconditioners or block operations, however, are defined for fine blocks (i.e., there are not two levels of local preconditioners).

It is apparent that the use of very small coarse blocks will degrade computing performance due to the overhead of procedure calls. Larger blocks can give better computational efficiency and convergence rate in preconditioned iterative methods, and computations with large dense blocks can be vectorized. In this article, we will rarely have need to mention fine blocks; thus, when we refer to “blocks” with no distinction, we normally mean coarse blocks.

To be concrete, we give an example of how a conversion routine is called when a block matrix is defined. The statement

```
BlockMat B("HBfile", 6, DENSE);
```

defines `B` to be a square block matrix where the blocks have dimension 6, and the blocks are stored in a format indicated by `DENSE` (which is of a C++ enumerated type). The other block type that is implemented is `CSR`, which stores blocks in the compressed sparse row format. The matrix is read from the file `HBfile`, which must be encoded in the standard Harwell-Boeing format [Duff et al. 1989]. (The dimension of the matrix does not need to be specified in the declaration since it is stored within the file.) To specify a *variable* block partitioning (with blocks with different sizes), other interfaces are available which use vectors to define the coarse and fine partitionings.

2.2 Specifying the preconditioning

A preconditioning for a block matrix is specified by choosing

(1) a *global preconditioner*, and
(2) a *local preconditioner* for each diagonal or pivot block to exactly or approximately invert the block or solve the corresponding set of equations.

For example, to fully define the conventional block Jacobi preconditioning, one must specify the global preconditioner to be block Jacobi and the local preconditioner to be LU factorization.

In addition, the block size of the matrix has a role in determining the effect of the preconditioning. At one extreme, if the block size is one, then the preconditioning is entirely determined by the global preconditioner. At the other extreme, if there is only one block, then the preconditioning is entirely determined by the local preconditioner. The block size parameterizes the effect and cost between the selected local and global preconditioners. The best method is likely to be somewhere between the two extremes.

For example, suppose symmetric successive overrelaxation (SSOR) is used as the global preconditioner, and complete LU factorization is used as the local preconditioner. For linear systems that are not too difficult to solve, SSOR may be used with a small block size. For more challenging systems, larger block sizes may be used, giving a better approximation to the original matrix. In the extreme, the matrix may be treated as a single block, and the method is equivalent to LU factorization.

A global preconditioner `M` is specified with a very simple form of declaration. In the case of block SSOR, the declaration is

```
BSSOR M;
```

Two functions are used to specify the local preconditioner and to provide parameters to the global preconditioner:

```
M.localprecon(LP_LU);       // LU factorization for the blocks
M.setup(B, 0.5, 3);         // BSSOR(omega=0.5, iterations=3)
```

Here `B` is the block matrix defined as in Section 2.1. The `setup` function provides the real data to the preconditioner, and performs all the computations necessary for setting up the global preconditioner, for example, the computation of the LU factors in this case. Therefore, `localprecon` must be called before `setup`. The `setup` function must be called again if the local preconditioner is changed. In these interfaces, the same local preconditioner is specified for all the diagonal blocks.

In general, however, the local preconditioners are not required to be the same. In some applications, different variables (e.g., velocity and pressure variables in a fluids simulation) may be blocked together. It may then make sense to write a specialized global preconditioner with an interface that allows different local preconditioners to be specified for each block.

2.2.1 *Global preconditioners.* The global preconditioners that we have implemented in BPKIT are listed in Table 1, along with the arguments of the `setup` function, and any default argument values. General reference works describing these global preconditioners and many of the local preconditioners described later are [Axelsson 1994; Barrett et al. 1994; Saad 1995]. See also the *BPKIT Reference Manual* [Chow and Heroux 1996]. Here we briefly specify these preconditioners and make a few comments on how they may be applied.

Table 1. Global preconditioners.

	`setup` arguments
`BJacobi`	none
`BSOR`	omega=1.0, iterations=1
`BSSOR`	omega=1.0, iterations=1
`BILUK`	level
`BTIF`	none

`BJacobi`, `BSOR` and `BSSOR` are block versions of the diagonal, successive overrelaxation, and symmetric successive overrelaxation preconditioners. `BILUK` is a block version of level-based incomplete LU (ILU) factorization. `BTIF` is an incomplete factorization for block tridiagonal matrices.

A preconditioner for a matrix A is often expressed as another matrix M which is somehow an approximation to A. However, M does not need to be explicitly formed, but instead, only the operation of M^{-1} on a vector is required. This operation is called the *preconditioning operation*, or the *application* of the preconditioner. For iterative methods based on biorthogonalization, the transposed preconditioning operator M^{-T} is also needed.

It is also possible to apply the preconditioner in a *split* fashion when the preconditioner has a factored form. For example, if M is factored as LU, then the preconditioned matrix is $L^{-1}AU^{-1}$, and the operations of L^{-1} and U^{-1} on a vector are required.

Many preconditioners M can be expressed in factored form. Consider the splitting of a block matrix A,

$$A = D_A - L_A - U_A$$

where D_A is the block diagonal of A, $-L_A$ is the strictly lower block triangular part, and $-U_A$ is the strictly upper part. The block SSOR preconditioner in the case of one iteration is defined by

$$M = \frac{1}{\omega(2-\omega)}(D_A - \omega L_A){D_A}^{-1}(D_A - \omega U_A).$$

The scale factor $1/\omega(2-\omega)$ is important if the iterative method is not scale invariant. When used as a preconditioner, the relaxation parameter ω is usually chosen to be 1, since selecting a value is difficult. However, if more than one iteration is used and the matrix is far from being symmetric and positive definite, underrelaxation may be necessary to prevent divergence. Also, the simpler block SOR preconditioner (with one iteration)

$$M = \frac{1}{\omega}(D_A - \omega L_A)$$

may be preferable over block SSOR if A is nonsymmetric. If k iterations of block SOR are used, the preconditioner has the form

$$M = \frac{1}{\omega}(D_A - \omega L_A)\left[\sum_{i=0}^{k-1}[(D_A - \omega L_A)^{-1}(\omega U_A + (1-\omega)D_A)]^i\right]^{-1}$$

although it is not implemented this way. Instead, the preconditioner is applied to a vector v by performing k SOR iterations on the system $Aw = v$ starting from the zero vector.

The level-0 block ILU preconditioner for certain structured matrices including block 5-point matrices can be written in a very similar form

$$M = (D - L_A)D^{-1}(D - U_A)$$

called the generalized block SSOR form. Here, D is the block diagonal matrix resulting from the incomplete factorization. In general, however, a level-based block ILU preconditioner is computed by performing Gaussian elimination and neglecting elements in the factors that fall out of a predetermined sparsity pattern. Level-based ILU preconditioners are much more accurate than relaxation preconditioners, but for general sparse matrices, have storage costs at least that of the original matrix.

Incomplete factorization of block tridiagonal matrices is popular for certain structured matrices where the blocks have banded structure. It is a special case of the generalized block SSOR form, and thus only a sequence of diagonal blocks needs to be computed and stored. The block partitioning may be along lines of a 2-D grid, or along planes of a 3-D grid. In general, any "striped" partitioning will yield a block tridiagonal matrix. The inverse-free form of block tridiagonal factorization is

$$M = (D^{-1} - L_A)(I - DU_A)$$

where D is a block diagonal matrix whose blocks D_i are defined by the recurrence

$$D_i = (A_{i,i} - A_{i,i-1}D_{i-1}A_{i-1,i})^{-1}$$

starting with $D_0 = 0$. This inverse-free form only requires matrix-vector multiplications in the preconditioning operation. However, the blocks are typically very large, and an approximate inverse is used in place of the exact inverse in the above equation to make the factorization *incomplete.* Many techniques for computing approximate inverses are available [Chow and Saad 1998].

2.2.2 *Local preconditioners.* Local preconditioners are either *explicit* or *implicit* depending on whether (approximate) inverses of blocks are explicitly formed. An example of an implicit local preconditioner is LU factorization.

The global preconditioners that involve incomplete factorization require the inverses of pivot blocks. For large block sizes, the use of approximate or exact *dense* inverses usually requires large amounts of storage and computation. Thus *sparse* approximate inverses should be used in these cases. Implicit local preconditioners produce inverses that are usually dense, and are therefore usually not computationally useful for block incomplete factorizations. This use of implicit local preconditioners is disallowed within BPKIT. We also apply this rule for small block sizes, since dense exact inverses are usually most efficient in these cases. (Note that the explicit local preconditioner `LP_INVERSE` for the `CSR` block type is meant to be used for testing purposes only. Also, if an exact factorization is sought, it is usually most efficient to use an LU factorization on the whole matrix.) The global preconditioners that involve block relaxation may use either explicit or implicit local preconditioners, but usually the implicit ones are used. Explicit local preconditioners can be appropriate for block relaxation when the blocks are small.

Local preconditioners are also differentiated by the type of the blocks on which they operate. Not all local preconditioners exist for all block types; incomplete factorization, for example, is only meaningful for sparse types. Thus, a local preconditioner must be chosen that matches the type of the block.

BPKIT requires the user to be aware of the restrictions in the above two paragraphs when selecting a local preconditioner. Due to the dynamic binding of C++ virtual functions, violations of these restrictions will only be detected at run-time.

Table 2 lists the local preconditioners that we have implemented, along with their `localprecon` arguments, their block types, and whether the local preconditioner is explicit or implicit. In contrast to the `setup` function, `localprecon` takes no default arguments. We have included an explicit exact inverse local preconditioner for the `CSR` format for comparison purposes (it would be inefficient to use it in block tridiagonal incomplete factorizations, for example).

Table 2. Local preconditioners.

	`localprecon` arguments	Block type	Expl./Impl.
`LP_LU`	none	`DENSE`	implicit
`LP_INVERSE`	none	`DENSE`	explicit
`LP_SVD`	alpha1, alpha2	`DENSE`	explicit
`LP_LU`	none	`CSR`	implicit
`LP_INVERSE`	none	`CSR`	explicit
`LP_RILUK`	level, omega	`CSR`	implicit
`LP_ILUT`	lfil, threshold	`CSR`	implicit
`LP_APINV_TRUNC`	semibw	`CSR`	explicit
`LP_APINV_BANDED`	semibw	`CSR`	explicit
`LP_APINV0`	none	`CSR`	explicit
`LP_APINVS`	lfil	`CSR`	explicit
`LP_DIAG`	none	`CSR`	explicit
`LP_TRIDIAG`	none	`CSR`	implicit
`LP_SOR`	omega, iterations	`CSR`	implicit
`LP_SSOR`	omega, iterations	`CSR`	implicit
`LP_GMRES`	restart, tolerance	`CSR`	implicit

`LP_LU` is an LU factorization with pivoting. `LP_INVERSE` is an exact inverse computed via LU factorization with pivoting. `LP_RILUK` is level-based relaxed incomplete LU factorization. `LP_ILUT` is a threshold-based ILU with control over the number of fill-ins [Saad 1994], which may be better for indefinite blocks. The local preconditions prefixed with `LP_APINV` are new approximate inverse techniques; see [Chow and Saad 1998] and [Chow and Heroux 1996] for details.

`LP_DIAG` is a diagonal approximation to the inverse, using the diagonal of the original block, and `LP_TRIDIAG` is a tridiagonal implicit approximation, ignoring all elements outside the tridiagonal band of the original block. `LP_SVD` uses the singular value decomposition $X = U\Sigma V^T$ to produce a dense approximate inverse $X^{-1} \approx V\bar{\Sigma}^{-1}U^T$, where $\bar{\Sigma}$ is Σ with its singular values thresholded by $\alpha_1\sigma_1 + \alpha_2$, a constant α_2 plus a factor α_1 of the largest singular value σ_1. This may produce a more stable incomplete factorization if there are many blocks to be inverted that are close to being singular [Yeremin 1995]. `LP_SOR`, `LP_SSOR` and `LP_GMRES` are iterative methods used as local preconditioners.

2.3 Interface with iterative methods

An object-oriented preconditioned iterative method requires that matrix and preconditioner objects define a small number of operations. In BPKIT, these operations are defined polymorphically, and are listed in Table 3.

For left and right preconditionings, the functions `apply` and `applyt` may be used to apply the preconditioning operator (M^{-1}, or its transpose) on a vector. *Split* (also called *two-sided,* or *symmetric*) preconditionings use `applyl` and `applyr` to apply the left and right parts of the split preconditioner, respectively. For an incomplete factorization $A \approx LU$, `applyl` is the L^{-1} operation, and `applyr` is the U^{-1} operation. To anticipate all possible functionality, the `applyc` function defines a combined matrix-preconditioner operator to be used, for example, to implement the Eisenstat trick [Eisenstat 1981]. If the Eisenstat trick is used with flexible preconditionings (described at the end of this section), the right preconditioner `apply` also needs to be used.

Two functions not listed here are matrix member functions that return the row and column dimensions of the matrix, which are useful for the iterative method code to help preallocate any work-space that is needed.

Not all the operations in Table 3 may be defined for all matrix and preconditioner objects, and many iterative methods do not require all these operations. The GMRES iterative method, for example, does not require the transposed operations, and the relaxation preconditioners usually do not define the split operations. This is a case where we violate an object-oriented programming paradigm, and give the parent classes all the specializations of their children (e.g., a specific preconditioner may not define `applyl` although the generic preconditioner does). This will be seen again in Section 3.2.

The argument lists for the functions in Table 3 use fundamental data types so that iterative methods codes are not forced to adopt any particular data structure for vectors. The interfaces use blocks of vectors to support iterative methods that use multiple right-hand sides. The implementation of these operations use Level 3 BLAS whenever possible. All the interfaces have the following form:

```
void mult(int nr, int nc, const double *u, int ldu, double* v, int ldv) const;
```

Table 3. Operations required by iterative methods.

Matrix operations	
`mult`	matrix-vector product
`trans_mult`	transposed matrix-vector product
Preconditioner operations	
`apply`	apply preconditioner
`applyt`	apply transposed preconditioner
`applyl`	apply left part of a split preconditioner
`applylt`	above, transposed
`applyr`	apply right part of a split preconditioner
`applyrt`	above, transposed
`applyc`	apply a combined matrix-preconditioner operator
`applyct`	above, transposed

where `nr` and `nc` are the row and column dimensions of the (input) blocks of vectors, `u` and `v` are arrays containing the values of the input and output vectors, respectively, and `ldu` and `ldv` are the leading dimensions of these respective arrays. The preconditioner operations are not defined as `const` functions, in case the preconditioner objects need to change their state as the iterations progress (and spectral information is revealed, for example).

When a non-constant operator is used in the preconditioning, a flexible iterative method such as FGMRES [Saad 1993] must be used. In BPKIT, this arises whenever GMRES is used as a local preconditioner. Users may wish to write advanced preconditioners that work *with* the iterative methods, and which change, for example, when there is a lack of convergence. This is a simple way of enhancing the robustness of iterative methods. In this case, the iterative method should be written as a class function whose class also provides information about convergence history and possibly approximate spectral information [Wu and Li 1995].

2.4 Fortran 77 interface

Many scientific computing users are unfamiliar with C++. It is usually possible, however, to provide an interface which is callable from any other language. BPKIT provides an object-oriented type of Fortran 77 interface. Objects can be created, and pointers to them are passed through functions as Fortran 77 integers. Consider the following code excerpt (most of the parameters are not important to this description):

```
call blockmatrix(bmat, n, a, ja, ia, num_block_rows, partit, btype)
call preconditioner(precon, bmat, BJacobi, 0.d0, 0.d0, LP_LU, 0.d0, 0.d0)
call flexgmres(bmat, sol, rhs, precon, 20, 600, 1.d-8)
```

The call to `blockmatrix` above creates a block matrix from the compressed sparse row data structure, given a number of arguments. This “wrapper” function is actually written in C++, but all its arguments are available to a Fortran 77 program. The integer `bmat` is actually a pointer to a block matrix object in C++. The Fortran 77 program is not meant to interpret this variable, but to pass it to other functions, such as `preconditioner` which defines a block preconditioner with a number of arguments, or `flexgmres` which solves a linear system using flexible GMRES. Similarly, `precon` is a pointer to a preconditioner object. The constant parameters `BJacobi` and `LP_LU` are used to specify a block Jacobi preconditioner, using LU factorization to solve with the diagonal blocks.

The matrix-vector product and preconditioner operations of Table 3 also have "wrapper" functions. This makes it possible to use BPKIT from an iterative solver written in Fortran 77. This was also another motivation to use fundamental types to specify vectors in the interface for operations such as `mult` (see Section 2.3).

Calling Fortran 77 from C++ is also possible, and this is done in BPKIT when it calls underlying libraries such as the BLAS. BPKIT illustrates how we were able to mix the use of different languages.

3. LOCAL MATRIX OBJECTS

A block matrix may contain blocks of more than one type. The best choice for the types of the blocks depends mostly on the structure of the matrix, but may also depend on the proposed algorithms and the computer architecture. For example, if a matrix has been reordered so that its diagonal blocks are all diagonal, then a diagonal storage scheme for the diagonal blocks is best. Inversion of these blocks would automatically use the appropriate algorithm. (The diagonal block type and the local preconditioners for it would have to be added by the user.)

To handle different block types the same way, instances of each type are implemented as C++ polymorphic objects (i.e., a set of related objects whose functions can be called without knowing the exact type of the object). The block types are derived from a *local matrix* class called `LocalMat`, a class that defines the common interface for all the block types. The global preconditioners refer to `LocalMat` objects. When `LocalMat` functions are called, the appropriate code is executed, depending on the actual type of the `LocalMat` object (e.g., `DENSE` or `CSR`).

In addition, each block type has a variety of local preconditioners. The explicitness or implicitness of local preconditioners need to be transparent, since, for example, either can be used in block SSOR. Thus both types of preconditioners are derived from the same base class. In particular, local preconditioners for a given block type are derived from the base class which is that block type (e.g., the `LP_SVD` local preconditioner for the `DENSE` type is derived from the `DENSE` block type). This gives the user the flexibility to treat explicit local preconditioners as regular blocks.

Implicit local preconditioners are not derived separately because logically they are related to explicit local preconditioners. All block operations that apply to explicit preconditioners also apply to local preconditioners; however, many of these operations are inefficient for local preconditioners, and their use has been disallowed to prevent improper usage. Implicit preconditioners cannot be derived separately from explicit preconditioners because of their similarity from the point of view of global preconditioners. The `LocalMat` hierarchy is illustrated in Figure 2, showing the derivation of block types and the subsequent derivation of local preconditioners.

These `LocalMat` classes form the "kernel" of BPKIT, and allow global preconditioners to be implemented without knowledge of the type of blocks or local preconditioners that are used. Users may also add to the kernel by deriving their own specific classes.

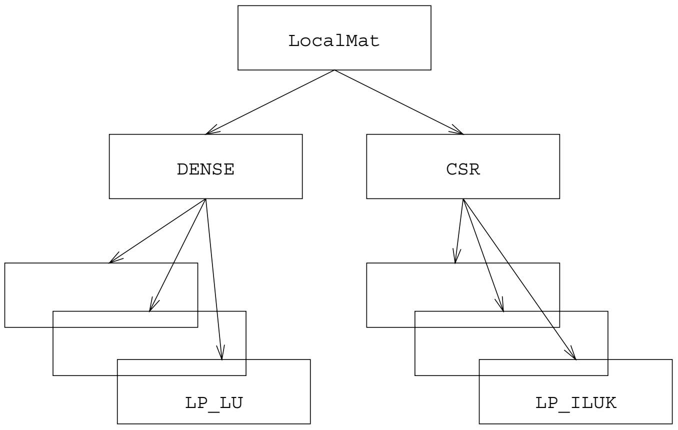

Fig. 2. `LocalMat` hierarchy.

The challenge of designing the `LocalMat` class was to determine what operations are required to implement block preconditioners and to give these operations semantics that allow an efficient implementation for all possible block types. The operations are implemented as C++ virtual functions. The following subsections describe these operations.

3.1 Allocating storage

An important difference between dense and sparse blocks is that the storage requirement for sparse blocks is not always known beforehand. Thus, in order to treat dense and sparse blocks the same way, storage is allocated for a block when it is required. As an optimization, if it is known that dense blocks are used (e.g., conversion of a sparse matrix to a block matrix with dense blocks), storage may be allocated beforehand by the user. Functions are provided to set the data pointers of the block objects. Thus it is possible to allocate contiguous storage for an *array* of dense blocks.

3.2 Local matrix functions

Table 4. Functions for `LocalMat` objects.

`B = A.CreateEmpty()`	$B = [\ \ ]$
`A.SetToZero(dim1,dim2)`	$A = 0$
`A.MatCopy(B)`	$A = B$
`B = A.CreateInv(lprecon)`	$B = \tilde{A}^{-1}$
`A.Mat_Trans(B)`	$B = A^T$
`A.Mat_Mat_Add(B, C, alpha)`	$C = A + \alpha B$
`A.Mat_Mat_Mult(B, C, alpha, beta)`	$C = \alpha AB + \beta C$
`A.Mat_Vec_Mult(b, c, alpha, beta)`	$c = \alpha Ab + \beta c$
`A.Mat_Trans_Vec_Mult(b, c, alpha, beta)`	$c = \alpha A^T b + \beta c$
`A.Mat_Vec_Solve(b, c)`	$c = A^{-1} b$
`A.Mat_Trans_Vec_Solve(b, c)`	$c = A^{-T} b$

Table 4 lists the functions that we have determined to be required for implementing the block preconditioners listed in Table 1. The functions are invoked by a block object represented by A. B and C are blocks of the same type as A, b and c are components from a block vector object, and α and β are scalars. The default value for α is 1 and for β is 0.

`CreateEmpty()` creates an empty block (0 by 0 dimensions) of the same class as that of `A`. This function is useful for constructing blocks in the preconditioner without knowing the types of blocks that are being used. `SetToZero(dim1, dim2)` sets `A` to zero, resetting its dimensions if necessary. This operation is not combined with `CreateEmpty()` because it is not always necessary to zero a block when creating it, and zeroing a block could be relatively expensive for some block types. `MatCopy(B)` copies its argument block to the invoking block. The original data held by the invoking block is released, and if the new block has a different size, the allocated space is resized. `CreateInv(lprecon)` provides a common interface for creating local preconditioners. `lprecon` is of a type that describes a local preconditioner with its arguments from Table 2. The exact or approximate inverse (explicit or implicit) of `A` is generated. The `CreateEmpty` and `CreateInv` functions create new objects (not just the real data space). These functions return pointers to the new objects to emphasize this point.

Overloading of the arithmetic operators such as `+` for blocks and local preconditioners has been sacrificed since chained operations such as $C = \alpha AB + \beta C$ would be inefficient if implemented as a sequence of elementary operations. In addition, these operators are difficult to implement without extra memory copying (for $A = B + C$, the $+$ operator will first store the result into a temporary before the result is copied into A by the $=$ operator).

These are the functions that we have found to be useful for block preconditioners. For example, $C = A + \alpha B$ is used in `BTIF`, $C = \alpha AB + \beta C$ is used in `BILUK`, and other functions are useful, for example, in matrix-vector product and triangular solve operations. Note in particular that `Mat_Trans_Mat_Mult` is not a useful function here, and has not been defined.

Note that local preconditioner objects also inherit these functions, although they do not need them all. For objects that are *implicit* local preconditioners, no matrix is formed, and operations such as addition (`Mat_Mat_Add`) do not make sense. For blocks for which no local preconditioner has been created, solving a system with that block (`Mat_Vec_Solve`) is not allowed. Here, again, we had to give the parent classes all the specializations of their derived classes. Table 5 indicates when the functions are allowed. An error condition is raised at run-time if the functions are used incorrectly.

Given these operations, a one-step block SOR code could be implemented as shown below. `Ap` is a pointer to a block matrix object which stores its block structure in `CSR` format (the `ia` array stores the block row pointers, and the `ja` array stores the block column indices). The pointers to the diagonal elements in `idiag` and the inverses of the diagonal elements `diag` were computed during the call to `setup`. `V` is a block vector object that allows blocks in a vector to be accessed as individual entries. The rest of the code is self-explanatory.

```
 1.    for (i=0; i<Ap->numrow(); i++)
```

Table 5. The types of objects that may be used with each function.

Function	Coarse blocks	Explicit local precon.	Implicit local precon.
`CreateEmpty`	*	*	
`SetToZero`	*	*	
`MatCopy`	*	*	
`CreateInv`	*	*	
`Mat_Trans`	*	*	
`Mat_Mat_Add`	*	*	
`Mat_Mat_Mult`	*	*	
`Mat_Vec_Mult`	*	*	
`Mat_Trans_Vec_Mult`	*	*	
`Mat_Vec_Solve`		*	*
`Mat_Trans_Vec_Solve`		*	*

```
    {
        for (j=ia[i]; j<idiag[i]; j++)
        {
            // V(i) = V(i) - omega * a[j] * V(ja[j])

            Ap->val(j).Mat_Vec_Mult(V(ja[j]), V(i), -omega, 1.0);
        }

        diag[i]->Mat_Vec_Solve(V(i), V(i));
    }
```

A block matrix that mixes different block types must be used very carefully. First, the restrictions for the different block types (Section 2.2.2) must not be violated. Second, unless we define arithmetic operations between blocks of different types, the incomplete factorization preconditioners cannot be used.

Our main design alternative was to create a block matrix class for each block type. The classes would be polymorphic and define a set of common operations that preconditioners may use to manipulate their blocks. A significant advantage of this design is that it is impossible to use local preconditioners of the wrong type (e.g., use incomplete factorization on a dense block). A disadvantage is that different block types (e.g., specialized types created for a particular application) cannot be used within the same block matrix.

Another alternative was to implement meta-matrices, i.e., blocks are nested recursively. It would be complicated, however, for users to specify these types of matrices and the levels of local preconditioners that could be used. In addition, there is very little need for such complexity in actual applications, and the two-level design (coarse and fine blocks) described in Section 2.1 should be sufficient.

4. NUMERICAL TESTS

The numerical tests were carried out on the matrices listed in Table 6. SHERMAN1 is a reservoir simulation matrix on a $10 \times 10 \times 10$ grid, with one unknown per grid point. This is a simple symmetric problem which we solve using partitioning by planes. WIGTO966 is from an Euler equation model and was supplied by Larry Wigton of Boeing. FIDAP019 models an axisymmetric 2-D developing pipe flow with the fully-coupled Navier-Stokes equations using the two-equation k-ϵ model for turbulence. The BARTHT1A and BARTHT2A matrices were supplied by Tim Barth of NASA Ames and are from a 2-D, high Reynolds number aerofoil problem, with a 1-equation turbulence model. The BARTHT2A model is solved with a preconditioner based on the less accurate but sparser BARTHT1A model.

Table 6. Test matrices, listed with their dimensions and numbers of nonzeros.

Matrix	n	no. nonz
SHERMAN1	1 000	3 750
WIGTO966	3 864	238 252
FIDAP019	12 005	259 879
BARTHT1A	14 075	481 125
BARTHT2A	14 075	1 311 725

Tables 7 and 9 show the results for SHERMAN1 with the block relaxation and incomplete factorization global preconditioners, using various local preconditioners. The arguments given for the global and local preconditioners in these tables correspond to those displayed in Tables 1 and 2 respectively. A block size of 100 was used. Since the matrix is block tridiagonal, `BILUK` and `BTIF` are equivalent. The tables show the number of steps of GMRES (FGMRES, if appropriate) that were required to reduce the residual norm by a factor of 10^{-8}. A dagger (†) is used to indicate that this was not achieved in 600 steps. Right preconditioning, 20 Krylov basis vectors and a zero initial guess were used. The right-hand side was provided with the matrix.

Since the local preconditioners have different costs, Tables 8 and 9 show the CPU timings (system and user times) for `BSSOR(1.,3)` and `BTIF`. The tests were run on one processor of a Sun Sparcstation 10. For this particular problem and choice of partitioning, the ILU local preconditioners required the least total CPU time with `BSSOR(1.,3)`. With `BTIF`, an exact solve was most efficient (i.e., the preconditioner was an exact solve).

Table 7. Number of GMRES steps for solving the SHERMAN1 problem with block relaxation global preconditioners and various local preconditioners.

	BJacobi	BSOR(1.,1)	BSOR(1.,3)	BSSOR(1.,1)	BSSOR(1.,3)
LP_INVERSE	88	40	17	24	13
LP_RILUK(0,0.)	93	71	53	402	48
LP_RILUK(1,0.)	89	43	24	41	20
LP_ILUT(2,0.)	85	63	46	319	44
LP_TRIDIAG	138	115	98	†	99
LP_SOR(1.,1)	†	541	†	†	491
LP_SSOR(1.,1)	508	408	395	†	411
LP_GMRES(150,0.1)	91	45	21	36	19
LP_APINVS(5)	108	70	57	274	56
LP_APINVO	171	130	122	†	121

Table 8. Number of GMRES steps and timings for solving the SHERMAN1 problem with `BSSOR(1.,3)` and various local preconditioners.

	`BSSOR(1.,3)`	CPU time (s)		
		precon	solve	total
`LP_INVERSE`	13	0.63	3.86	4.49
`LP_RILUK(0,0.)`	48	0.01	1.86	1.87
`LP_RILUK(1,0.)`	20	0.02	0.87	0.89
`LP_ILUT(2,0.)`	44	0.03	1.73	1.76
`LP_TRIDIAG`	99	0.01	3.87	3.88
`LP_SOR(1.,1)`	491	0.00	20.80	20.80
`LP_SSOR(1.,1)`	411	0.00	19.40	19.40
`LP_GMRES(150,0.1)`	19	0.00	29.23	29.23
`LP_APINVS(5)`	56	0.27	2.48	2.75
`LP_APINV0`	121	0.51	5.21	5.72

Table 9. Number of GMRES steps and timings for solving the SHERMAN1 problem with block incomplete factorization and various local preconditioners.

	`BTIF`	CPU time (s)		
		precon	solve	total
`LP_INVERSE`	1	1.44	0.15	1.59
`LP_DIAG`	†	0.01	†	†
`LP_APINV0`	123	0.52	3.73	4.25
`LP_APINVS(5)`	64	0.32	1.98	2.30
`LP_APINVS(10)`	33	1.14	1.08	2.22

Tables 10 and 11 show the number of GMRES steps for the BARTHT2A matrix. A random right-hand side was used, and the initial guess was zero. The GMRES tolerance was 10^{-8} and 50 Krylov basis vectors were used. In Table 10, block incomplete factorization was used as the global preconditioner, and LU factorization was used as the local preconditioner. In Table 11, block SSOR with one iteration and $\omega = 1$ was used as the global preconditioner, and level-3 ILU was used as the local preconditioner.

Table 10. Number of GMRES steps for solving the BARTHT2A problem with `BILUK-LP_LU`.

block	`BILUK` level		
size	0	1	2
5	436	183	130
10	184	118	95
15	141	100	94

Tables 12 and 13 show the results for WIGTO996 using block incomplete factorization. The right-hand side was the vector of all ones, and the GMRES tolerance was 10^{-8}. The other parameters were the same as those in the previous experiment. The failures in Table 12 are due to inaccuracy for low fill levels, and instability for high levels. In Table 13, `LP_SVD(0.1,0.)` used as the local preconditioner gave the

Table 11. Number of GMRES steps for solving the BARTHT2A problem with `BSSOR(1.,1)-LP_RILUK(3,0.)`.

block size	GMRES steps
60	266
120	273
240	210

best results. `LP_SVD(0.1,0.)` indicates that the singular values of the pivot blocks were thresholded at 0.1 times the largest singular value.

Table 12. Number of GMRES steps for solving the WIGTO966 problem with `BILUK-LP_INVERSE`.

block	`BILUK` level			
size	0	1	2	3
4	†	†	†	†
8	†	†	94	75
16	†	77	400	†

Table 13. Number of GMRES steps for solving the WIGTO966 problem with `BILUK-LP_SVD(0.1,0.)`.

block	`BILUK` level			
size	0	1	2	3
4	50	42	39	35
8	44	35	32	30
16	40	36	32	30

Now we show some results with block tridiagonal incomplete factorization preconditioners using general sparse approximate inverses. The matrix FIDAP019 was partitioned into a block tridiagonal system using a constant block size of 161 (the last block has size 91). Since the matrix arises from a finite element problem, a more careful selection of the partitioning could have yielded better results.

The rows of the system were scaled by their 2-norms, and then their columns were scaled similarly, since the matrix contains different equations and variables. A Krylov subspace size of 50 for GMRES was used. The right-hand side was constructed so that the solution is the vector of all ones. We compare the result with the pair of global-local preconditioners `BILUK(0)-LP_SVD(0.5,0.)`, using a block size of 5 (`LP_SVD(0.5,0.)` gave the best result after several trials). Table 14 shows the number of GMRES steps to convergence, timings for setting up the preconditioner and for the iterations, and the number of nonzeros in the preconditioner. The experiments were carried out on one processor of a Sun Sparcstation 10.

The timings show that some combinations of the `BTIF` global preconditioner with the `APINVS` local preconditioner are comparable to `BILUK(0)-LP_SVD(0.5,0.)`, but use much less memory, since only the approximate inverses of the pivot blocks need to be stored. Although the actual number of nonzeros in the matrix is 259 879, there were 39 355 block nonzeros required for `BILUK`, and therefore almost a million

Table 14. Test results for the FIDAP019 problem.

	GMRES	CPU time (s)			nonzeros
	steps	precon	solve	total	in precon
`BILUK(0)-LP_SVD(0.5,0.)`	87	15.98	143.18	159.16	983 875
`BTIF-LP_APINVS(10)`	186	56.20	113.41	169.61	120 050
`BTIF-LP_APINVS(5)`	328	44.58	186.34	230.92	60 025

entries which were needed to be stored. The `APINVS` method produced approximate inverses that were sparser than the original pivot blocks. See [Chow and Saad 1998] for more details.

There is often heated debate over the use of C++ in scientific computing. Ideally, C++ and Fortran 77 programs that are coded similarly should perform similarly. However, by using object-oriented features in C++ to make a program more flexible and maintainable, researchers usually encounter a 10 to 30 percent performance penalty [Jiang and Forsyth 1995]. If optimized kernels such as the BLAS are called, then the C++ performance penalty can be very small for large problems, as a larger fraction of the time is spent in the kernels.

Since C++ and Fortran 77 programs will usually be coded differently, a practical comparison is made when a general code such as BPKIT is compared to a *specialized* Fortran 77 code. Here we compare BPKIT to an optimized block SSOR preconditioner with a GMRES accelerator. This code performs block relaxations of the form

$$\begin{aligned} \delta &\Leftarrow A_{ii}^{-1} r_i \\ x_i &\Leftarrow x_i + \delta \\ r &\Leftarrow r - A_{:,i}\delta \end{aligned}$$

for a block row i, where A_{ii} is the i-th diagonal block of A, $A_{:,i}$ is the i-th block column of A, x_i is the i-th block of the current solution, and r is the current residual vector. Notice that the update of the residual vector is very fast if A is stored by sparse columns and not by blocks. Since BPKIT stores the matrix A by blocks for flexibility, it is interesting to see what the performance penalty would be for this case.

Tables 15 and 16 show the timings for block SSOR on a Sun Sparcstation 10 and a Cray C90 supercomputer, for the WIGTO966 matrix. In this case, the right-hand side was constructed so that the solution is a vector of all ones; the other parameters were the same as before. All programs were optimized at the highest optimization level; `clock` was used to measure CPU time (user and system) for the C++ programs, and `etime` and `timef` were used to measure the times for the Fortran 77 programs on the Sun and Cray computers, respectively. One step of block SSOR with $\omega = 0.5$ was used in the tests. The local preconditioner was an exact LU factorization. Results are shown for a large range of block sizes, and in the case of BPKIT, for both `DENSE` and `CSR` storage schemes for the blocks. The last column of each table gives the average time to perform one iteration of GMRES.

The results show that the specialized Fortran 77 code has better performance over a wide range of block sizes. This is expected because the update of the residual, which is the most major computation, is not affected by the blocking.

If dense blocks are used, BPKIT can be competitive on the Cray by using large block sizes, such as 128. Blocks of this size contain many zero entries which are treated as general nonzero entries when a dense storage scheme is used. However, vectorization on the Cray makes operations with large dense blocks much more efficient.

If sparse blocks are used, BPKIT can be competitive on the workstation with moderate block sizes of 8 or 16. Operations with smaller sparse blocks are inefficient, while larger blocks imply larger LU factorizations for the local preconditioner.

This comparison using block SSOR is dramatic since two very different data structures are used. Comparisons of level-based block ILU in C++ and Fortran 77 show very small differences in performance, since the data structures used are similar [Jiang and Forsyth 1995].

In conclusion, the types and sizes of blocks must be chosen carefully in BPKIT to attain high performance on a particular machine. The types and sizes of blocks should also be chosen in conjunction with the requirements of the preconditioning algorithm and the block structure of the matrix. Based on the above experiments, Table 17 gives an idea of the approximate block sizes that should be used for BPKIT, given no other constraints.

5. CONCLUDING REMARKS

This article has described an object-oriented framework for block preconditioning. Polymorphism was used to handle different block types and different local preconditioners. Block types and local preconditioners form a "kernel" on which the block preconditioners are built. Block preconditioners are written in a syntax comparable to that for non-block preconditioners, and they work for matrices containing any block type. BPKIT is easily extensible, as an object-oriented code would allow. We have distinguished between explicit and implicit local preconditioners, and deduced the operations and semantics that are useful for polymorphically manipulating blocks. Timings against a specialized and optimized Fortran 77 code on both workstations and Cray supercomputers show that this framework can approach the efficiency of such a code, as long as suitable block sizes and block types are chosen. We believe we have found a suitable compromise between Fortran 77-like performance and C++ flexibility. A significant contribution of BPKIT is the collection of high-quality preconditioners under a common, concise interface.

Block preconditioners can be more efficient and more robust than their non-block counterparts. The block size parameterizes between a local and global method, and is valuable for compromising between accuracy and cost, or combining the effect of two methods. The combination of local and global preconditioners leads to a variety of useful methods, all of which may be applicable in different circumstances.

ACKNOWLEDGMENTS

We wish to thank Yousef Saad, Kesheng Wu and Andrew Chapman for their codes and for helpful discussions. We also wish to thank Larry Wigton and Tim Barth for providing some of the test matrices, and Tim Peck for helping us with editing. This article has benefited substantially from the comments and suggestions of one of the anonymous referees, and we are grateful for his time and patience.

Table 15. WIGTO966: `BSSOR(0.5,1)-LP_LU`, Sun Sparc 10 timings.

Specialized Fortran 77 program					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	1.87	193.09	194.96	0.3899
8	240	1.21	91.47	92.68	0.3862
16	306	1.04	118.78	119.82	0.3916
32	300	1.21	124.41	125.63	0.4188
64	221	1.79	103.85	105.65	0.4781
128	212	3.86	124.15	128.02	0.6039

BPKIT, dense blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.25	305.97	306.22	0.6124
8	240	0.25	119.16	119.41	0.4975
16	306	0.38	193.97	194.35	0.6351
32	300	0.73	303.10	303.83	1.0128
64	221	1.27	376.02	377.29	1.7072
128	212	3.66	559.05	562.71	2.6543

BPKIT, sparse blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.24	284.69	284.93	0.5699
8	240	0.34	106.67	107.01	0.4459
16	306	0.62	129.96	130.58	0.4267
32	300	1.06	137.50	138.56	0.4619
64	221	1.87	123.34	125.21	0.5666
128	212	4.42	162.58	167.00	0.7877

Table 16. WIGTO966: `BSSOR(0.5,1)-LP_LU`, Cray C90 timings.

Specialized Fortran 77 program					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.075	13.92	14.00	0.0280
8	240	0.065	5.57	5.64	0.0235
16	306	0.065	7.02	7.09	0.0232
32	300	0.089	7.00	7.09	0.0237
64	221	0.140	6.03	6.17	0.0279
128	212	0.296	7.23	7.53	0.0355

BPKIT, dense blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.115	183.92	184.04	0.3681
8	240	0.080	36.07	36.15	0.1506
16	306	0.072	26.75	26.83	0.0877
32	300	0.079	16.79	16.87	0.0563
64	221	0.127	8.24	8.37	0.0379
128	212	0.290	6.73	7.03	0.0332

BPKIT, sparse blocks					
block	GMRES	time (s)			
size	steps	precon	solve	total	average
4	500	0.27	282.32	282.59	0.5652
8	240	0.39	92.27	92.67	0.3861
16	306	0.83	106.72	107.56	0.3515
32	300	1.38	110.42	111.80	0.3727
64	221	2.44	99.15	101.59	0.4597
128	212	5.39	132.92	138.31	0.6524

Table 17. Recommended block sizes.

Block type	Sun	Cray
`DENSE`	8	128
`CSR`	16	16

REFERENCES

Axelsson, O. 1994. *Iterative Solution Methods.* Cambridge University Press, Cambridge.

Axelsson, O., Brinkkemper, S., and Il'in, V. P. 1984. On some versions of incomplete block-matrix factorization iterative methods. *Lin. Alg. Appl. 58*, 3–15.

Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* 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*, 50–80.

Carney, S., Heroux, M. A., Li, G., and Wu, K. 1994. A revised proposal for a sparse BLAS toolkit. Technical Report 94-034, Army High Performance Computing Research Center, Minneapolis, MN.

Chow, E. and Heroux, M. A. 1996. BPKIT Block preconditioning toolkit. Technical Report UMSI 96/183, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN.

Chow, E. and Saad, Y. 1997. Approximate inverse techniques for block-partitioned matrices. *SIAM J. Sci. Comput. 18*, to appear.

Chow, E. and Saad, Y. 1998. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput. 19*, to appear.

Clay, R. L. 1997. ISIS++: iterative scalable implicit solver (in C++). Sandia National Laboratories, Livermore, CA.

Concus, P., Golub, G. H., and Meurant, G. 1985. Block preconditioning for the conjugate gradient method. *SIAM J. Sci. Statist. Comput. 6*, 309–332.

Demmel, J. 1989. LAPACK: A portable linear algebra library for supercomputers. In *Proc. 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design* (December 1989).

Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1–17.

Dongarra, J. J., Lumsdaine, A., Niu, X., Pozo, R., and Remington, K. 1994. A sparse matrix library in C++ for high performance architectures. In *Proc. Object Oriented Numerics Conference* (Sun River, OR, 1994).

Dongarra, J. J., Pozo, R., and Walker, D. W. 1993. An object oriented design for high performance linear algebra on distributed memory architectures. In *Proc. Object Oriented Numerics Conference* (Snowbird, CO, 1993).

Duff, I. S., Grimes, R. G., and Lewis, J. G. 1989. Sparse matrix test problems. *ACM Trans. Math. Softw. 15*, 1–14.

Eijkhout, V. 1996. ParPre: A parallel preconditioners package. Manuscript.

Eisenstat, S. C. 1981. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Statist. Comput. 2*, 1–4.

Fan, Q., Forsyth, P. A., McMacken, J. R. F., and Tang, W.-P. 1996. Performance issues for iterative solvers in device simulation. *SIAM J. Sci. Comput. 17*, 100–117.

Jiang, H. and Forsyth, P. A. 1995. Robust linear and nonlinear strategies for solution of the transonic Euler equations. *Computers & Fluids 24*, 753–770.

Jones, M. T. and Plassmann, P. E. 1995. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, Argonne, IL.

Kolotilina, L. Yu., Kaporin, I. E., and Yeremin, A. Yu. 1991. Block SSOR preconditionings for high-order 3D FE systems. Incomplete BSSOR preconditionings. *Lin. Alg. Appl. 154–156*, 647–674.

Kolotilina, L. Yu. and Yeremin, A. Yu. 1986. On a family of two-level preconditionings of the incomplete block factorization type. *Soviet J. Numer. Anal. Math. Model. 1*, 293–320.

Machiels, L. and Deville, M. O. 1997. Fortran 90: An entry to object-oriented programming for solution of partial differential equations. *ACM Trans. Math. Softw. 23*, 32–49.

Remington, K. A. and Pozo, R. 1996. NIST sparse BLAS user's guide. Technical report, National Institute of Standards and Technology, Gaithersburg, Maryland.

Saad, Y. 1990. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA.

Saad, Y. 1993. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Statist. Comput. 14*, 461–469.

Saad, Y. 1994. ILUT: A dual threshold incomplete ILU factorization. *Num. Lin. Alg. Appl. 1*, 387–402.

Saad, Y. 1995. *Iterative Methods for Sparse Linear Systems.* PWS Publishing, New York.

Smith, B., Gropp, W., and McInnes, L. C. 1995. PETSc 2.0 users' manual. Technical Report ANL-95/11, Argonne National Laboratory, Argonne, IL.

Stroustrup, B. 1991. *The C++ Programming Language* (2 ed.). Addison-Wesley, Reading, MA.

Underwood, R. R. 1976. An approximate factorization procedure based on the block Cholesky decomposition and its use with the conjugate gradient method. Technical Report NEDO-11386, General Electric Co., Nuclear Energy Div., San Jose, CA.

Wu, K. and Li, G. 1995. BKAT: An object-oriented block Krylov accelerator toolkit. Presentation at Cray Research, Inc. Available from `kewu@kjwu.lbl.gov`.

Yeremin, A. Yu. 1995. Private communication.