Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

---

ANL-95/48

---

# BlockSolve95 Users Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems

by

MARK T. JONES* AND PAUL E. PLASSMANN

Mathematics and Computer Science Division

December 1995
(Revised June 1997)

*Address: Dept. of Computer Science, The University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The first author was supported in part by a 1994–95 University of Tennessee Professional Development award and NSF Grants ASC-9501583 and ASC-9411394.

# Contents

# BlockSolve95 Users Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems

by

Mark T. Jones and Paul E. Plassmann

**Abstract**

*BlockSolve95* is a software library for solving large, sparse systems of linear equations on massively parallel computers or networks of workstations. The matrices must be symmetric in structure; however, the matrix nonzero values may be either symmetric or nonsymmetric. The nonzeros must be real valued. *BlockSolve95* uses a message-passing paradigm and achieves portability through the use of the MPI message-passing standard. Emphasis has been placed on achieving both good processor performance through the use of higher-level BLAS and scalability through the use of advanced algorithms. This report gives detailed instructions on the use of *BlockSolve95* and descriptions of a number of program examples that can be used as templates for application programs.

# 1 Introduction

*BlockSolve95* is a scalable parallel software library primarily intended for the solution of sparse linear systems that arise from physical models, especially problems involving multiple degrees of freedom at each node. For example, when the finite element method is used to solve practical problems in structural engineering, each node typically has two to five degrees of freedom; *BlockSolve95* is designed to take advantage of problems with this type of local structure. *BlockSolve95* is also reasonably efficient for problems that have only one degree of freedom associated with each node, such as the three-dimensional Poisson problem. *BlockSolve95* is general purpose; we require only that the matrices be sparse and symmetric in structure (but not necessarily in value).[1]

*BlockSolve95* has incorporated several features that allow efficient performance on diverse parallel architectures. We summarize these features below:

- Every aspect of the linear system solution—the computation of a matrix ordering, the renumbering, the computation of the preconditioner, and the iterative solution—is done in parallel.
- *BlockSolve95* runs on a variety of parallel architectures and can easily be ported to others. The MPI message-passing standard [4, 13] is used to achieve portability across architectures. Machines on which *BlockSolve95* has been tested include the IBM SP series, the Cray T3D and T3E, the SGI Cray Origin2000, the HP-Convex Exemplar, the Intel Paragon, and networks of Sun, SGI, DEC alpha, and HP workstations. The `bmake` directory contains contains a complete set of the machine-dependent makefiles for the architectures currently supported.
- The software uses an efficient implementation of the parallel coloring algorithm described in [9] that allows for the efficient computation of matrix orderings and the scalable performance of the linear solver described below.
- The software is designed to solve linear systems whose sizes are the same order as the total amount of memory available on a parallel machine. For example, Table 1 shows how effectively *BlockSolve95* uses the available memory on two different architectures for the high-temperature superconductor modeling application described in [8]. Note that some of each processor's memory is used by the operating system and program executable (e.g., approximately 4 MBytes and 1 MByte, respectively, on the Intel DELTA). For this application an unpermuted version of the matrix is maintained in addition to the memory required by *BlockSolve95*. Moreover, the memory required to store *only* the indices and the double-precision values for $1.0 \times 10^8$ nonzeros is 1.2 GBytes.

Table 1: Effective use of total memory by *BlockSolve95*

| Machine | Number of Processors | Number of Matrix Nonzeros | Memory per Processor (Usable Memory) | Total Memory (Usable Memory) |
|---|---|---|---|---|
| Intel DELTA | 512 | $1.9 \times 10^8$ | 16 MBytes (11 MBytes) | 8 GBytes (5.6 MBytes) |
| IBM SP1 | 128 | $5.1 \times 10^8$ | 128 MBytes | 16 GBytes |

[1]Whenever we use the term *nonsymmetric* with respect to *BlockSolve95*, we mean symmetric in structure and nonsymmetric in value. Thus, *BlockSolve95* handles most matrices arising in finite-element calculations; however, it cannot handle general nonsymmetric matrices.

- The software is designed to achieve scalable performance. With a constant number of unknowns per processor, a matrix nonzero structure resulting from local graph connections (the usual case with any finite-element mesh), and a reasonable partitioning of vertices to processors, the average processor performance is roughly constant. For example, for the piezoelectric crystal modeling application described in [7], the average processor performance on the Intel DELTA varied from 4.16 MFlops to 3.83 MFlops when the number of processors was increased from 128 to 512.
- The software is designed to use Level 2 and Level 3 dense BLAS (the Basic Linear Algebra Subroutines) to achieve efficient use of processors on different architectures. For example, Table 2 shows the average processor performances for the application described in [8]. Note that a reasonable percentage of the LINPACK benchmark performance is achieved for a *sparse* matrix calculation on these very different processor architectures.

Table 2: Efficient use of different processor architectures by *BlockSolve95*

| Machine | Processor Architecture | Number of Processors | Avg. Proc. Performance Achieved | Total Performance |
|---|---|---|---|---|
| Intel DELTA | Intel i860 | 512 | 8.3 MFlops | 4.26 GFlops |
| IBM SP1 | RS/6000-370 | 128 | 20.5 MFlops | 2.62 GFlops |

- The software is a general-purpose sparse solver. Hence, *BlockSolve95* is effective also on problems arising from unstructured meshes. For example, a total performance of 2.2 GFlops was achieved on 512 processors of the Intel DELTA for an adaptive mesh calculation using high-order shell elements in a three-dimensional geometry [10].
- The software requires minimal input. You must give *BlockSolve95* the matrix nonzeros and global indices of rows corresponding to the unknowns assigned to each processor and the mapping functions to translate between global and local indexing.
- *BlockSolve95* is designed to be most effective within real application codes. In our experience, most application codes must solve the same linear systems with several different right-hand sides and/or solve linear systems with the same structure, but different matrix values, multiple times. *BlockSolve95* has, therefore, been designed to work well in this situation.

The remainder of this manual is organized as follows. We begin in §2 with a brief description of the algorithms used in *BlockSolve95.* In §3, §4, and §5 we present the routines required to set up the context and matrix data structures and solve linear systems with *BlockSolve95.* In §6 we discuss a number of details dealing with flop counting, BLAS performance, message-passing options, and other issues that arise in tuning the performance of an application. The installation and availability of *BlockSolve95* are discussed in §7, and in §8 we describe the program examples that are included with this version. Finally, in §9 we discuss related software and plans for future versions of the code.

This document is intended to be used primarily as a reference. If you are a new user, we recommend using the *BlockSolve95* examples as templates for your application codes.

## 2 Algorithm Descriptions

The primary algorithmic components of *BlockSolve95* are

- the computation of a matrix ordering allowing for the scalable inversion of the triangular systems arising from incomplete matrix factorizations;
- the automatic extraction of clique and identical node (i-node) information from unstructured systems that allows for the use of higher-level BLAS; and
- the computation of incomplete sparse matrix factorizations for use as preconditioners in the iterative solution of linear systems.

The matrix-ordering algorithms are used to ensure that, for many problems, the performance per processor remains roughly constant as the problem size and number of processors are increased. Computational efficiency is obtained through the use of higher-level BLAS, efficient matrix storage schemes, and precomputed communication data structures. For symmetric matrices, incomplete Cholesky factorizations may be computed; and for general matrices, incomplete LU factorizations may be obtained. These preconditioners are designed so that their application is both efficient and as scalable as possible.

In addition, *BlockSolve95* contains implementations of several well-known iterative methods. A preconditioned conjugate gradient method can be used for symmetric, positive definite systems. For symmetric indefinite systems, the preconditioned SYMMLQ algorithm may be used.[2] For nonsymmetric systems, the GMRES method may be used. In addition, the PETSc software package [1] contains an interface to the *BlockSolve95* preconditioners; this interface may be used if other iterative methods are required. For basic information on iterative methods, see [2].

You can select from a number of different preconditioners. The first is a simple diagonal scaling of the matrix, which can be used by itself or in conjunction with another preconditioner. The other preconditioning options are incomplete Cholesky or LU factorization, SSOR ($\omega = 1$), and block Jacobi (where the blocks are the cliques of the graph associated with the sparse matrix). Perhaps the most generally applicable selection is the incomplete factorization with diagonal scaling.[3] The incomplete factorization is the algorithm for which *BlockSolve95* was designed; this approach has proved useful in a wide variety of practical problems.

*BlockSolve95* does not partition a matrix across the processors. Instead, *BlockSolve95* assumes that the given partitioning is a good one. As such, its performance may be limited by the quality of the partitioning. We assume that the right-hand side and the solution vector are partitioned in the same manner as the rows of the sparse matrix. See [5] for more information on partitioning heuristics.

We achieve parallelism in the conjugate gradient, SYMMLQ, and GMRES implementations by partitioning the vectors used in these algorithms in the same manner that the rows of the matrix are partitioned across the processors. For these algorithms it is (for the most part) a simple matter of executing inner products and `daxpy` operations in parallel.

### 2.1 Processor Performance

To achieve good performance on each node, we reorder the matrix and use a layered data structure to allow the use of the higher-level dense BLAS. This reordering is particularly important

[2]The SYMMLQ algorithm requires a positive definite preconditioner, and this requirement can be a serious limitation if the matrix being solved is very indefinite. By "very indefinite," we mean that the matrix has many negative and many positive eigenvalues.

[3]Two possible exceptions to this recommendation are (1) if the matrix has no or very small cliques and identical nodes (in which case the factorization may be very slow) and (2) if the space for the incomplete factorization is not available.

on machines that use high-performance RISC chips on which good performance can be achieved only by using such operations. The reordering of the matrices is based on the identification of two structures that commonly arise in the graph associated with the matrix nonzeros: identical nodes and cliques.

Identical nodes (i-nodes) typically exist when multiple degrees of freedom are associated with each vertex in the graph. Cliques are found in many graphs associated with sparse matrices, but cliques typically are found in graphs where multiple degrees of freedom are associated with each vertex and the local connectivity of the graph is large. For example, if one uses a second-order, three-dimensional finite element in a typical structural engineering problem (with three degrees of freedom per vertex), clique sizes of up to 81 can be found. In general, the larger the cliques or identical nodes, the better the performance. This technique has been used with great success in direct matrix factorization methods.

We illustrate these structures in Figure 1 where we depict a subsection of a graph that would arise from a two-dimensional, bilinear, multicomponent finite-element model with three degrees of freedom per discretization point. We illustrate the three degrees of freedom by the three dots at each node point; the linear, quadrilateral elements imply that the twelve degrees of freedom sharing the four node points of each face are completely connected. In the figure we show edges only between the nodes, these edges represent the complete interconnection of all the vertices on each element or face.

The dashed lines in the figure represent a geometric partitioning of the grid; we assume that the vertices in the central region are all assigned to one processor. We note that the adjacency structures of the vertices at the same geometric node (i.e., the nonzero structure of the associated variables) are identical. *BlockSolve95* takes advantage of these so-called i-nodes by maintaining only one copy of the indexing for each set of identical rows and by storing nonzeros for these rows in a dense matrix form that can be used with the BLAS.

A second data structure layer is determined by the *clique* structure of the graph. Recall that a clique is a completely connected subgraph. In the upper right of Figure 1, the grouping of vertices by the dotted lines partitions the graph into cliques. *BlockSolve95* associates a "supernode" with each clique and orders the unknowns associated with each clique consecutively. Based on this ordering, the submatrix corresponding to the unknowns in the same clique corresponds to a dense matrix. In the sparse matrix, these dense matrices manifest themselves as dense blocks on the matrix diagonal. The *BlockSolve95* data structures explicitly store these dense blocks to allow for greater efficiency in using the BLAS.

## 2.2 Multicolor Ordering

Following the graph reductions that are accomplished after the identification of i-nodes and cliques in the sparse matrix structure, *BlockSolve95* must construct a matrix ordering based on the reduced graph, illustrated at the bottom of Figure 1. To achieve scalable parallel performance in the incomplete factorization and SSOR preconditioners, *BlockSolve95* colors the reduced graph using a parallel coloring heuristic [9]. The graph reduction and coloring phases of the computation are efficient and typically require a small amount of time relative to the incomplete matrix factorization routine.

The advantage of using a graph coloring is that the number of colors required is essentially a *local* property of the graph; thus, the number of colors is roughly independent of the number processors used for a fixed discretization scheme. This fact allows for the scalable performance of the *BlockSolve95* package [11]. The trade-off for this scalability is that these multicolor orderings may not be the optimal orderings to choose for minimizing the number of iterations required for the iterative solver; the convergence of the iterative solver with respect to different orderings is highly problem dependent. However, the combination of coloring a general symmetric sparse matrix and the incomplete Cholesky algorithm has proved to be successful for solving large

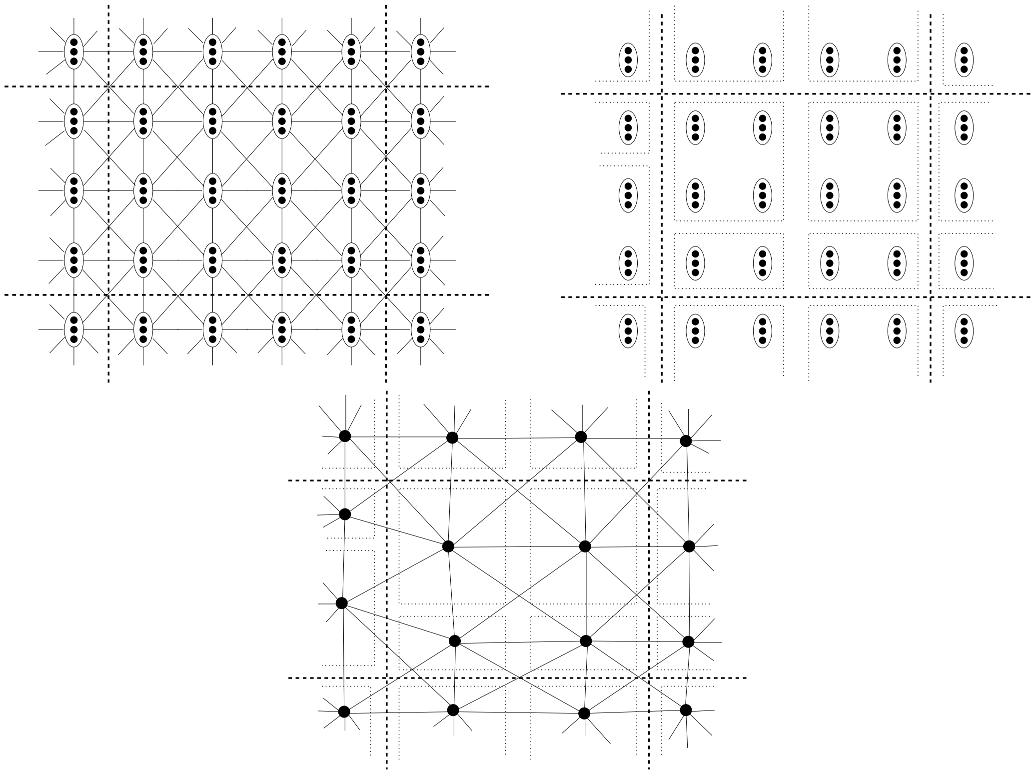

Figure 1: A subgraph generated by a two-dimensional, bilinear finite-element model with three degrees of freedom per discretization point. The geometric partition shown by the dotted lines yields an assignment of the vertices in the enclosed subregion to one processor. The upper right figure shows one possible decomposition of the graph into local cliques, and the lower figure shows the corresponding clique (or supernode) graph.

problems on scalable parallel architectures [7]. In [6] we address the issue of convergence of this combination of algorithms for the model problem.

### 2.3 Communication Efficiency

The computational kernel of the *BlockSolve95* package is the solution of triangular matrix systems and triangular matrix multiplication. Triangular matrices of the same structure are solved and multiplied in parallel many times in the use of one of the iterative solvers. The key observation is that the required interprocessor communication in triangular system solve or multiply is the same every time; the only difference is in the floating-point values that are sent and received.

To take advantage of this repeated communication pattern, the *BlockSolve95* routines in `BMcomp_msg()` are used to build reusable data structures. During each communication phase each processor needs to send and receive data from some subset of the processors (a small subset, if the partitioning is good). Each message to be sent is an array of floating-point values and an array of integer values (matrix indices). A key observation is that these integer values do not change; only the floating-point values change.

Prior to any triangular matrix solution and multiplication, each processor declares (via function calls in `BMcomp_msg()`) the messages that it will be sending and receiving and specifies the integer index values associated with these messages. The routines in `BMcomp_msg()` build data structures by assembling the declarations from each processor. In addition, the integer index values associated with the messages are sent to the appropriate processor and stored there; these values will not be sent in the future. Since the size of the required messages is now known, message buffers can be preallocated.

We currently perform no special communication pattern optimizations in `BMcomp_msg()` although these routines have been set up for this possibility. The optimizations currently performed are

1. presending of integer values;
2. preallocation of message buffers, allowing "forced" messages; `MPI_Irsend()`, to be used; and
3. no requests are made between processors for non-local data (processors use local information to know what messages to expect and what to send).

For an example of the message setup process, see the routine `BSsetup_forward()`. For an example of how the message data structures are used, see the routine `BSforward()`.

# 3 The *BlockSolve95* Computing Environment

In this section we describe the *BlockSolve95* computing environment. *BlockSolve95* has an initialization routine that must be called prior to any other *BlockSolve95* routine and a finalization routine that must be called last. We also introduce the most important *BlockSolve95* data structure—`BSprocinfo`—the *BlockSolve95* context. All information about the parallel environment, preconditioner options, and settings for the iterative solvers is communicated to the *BlockSolve95* package through the `BSprocinfo` context data structure.

## 3.1 Initializing and Finalizing the *BlockSolve95* Environment

The first call to a *BlockSolve95* routine must be the initialization routine `BSinit()`. The definition and arguments for this routine are as follows.

```
int BSinit(int *argc, char ***args);
```

The arguments `argc` and `argv` should be the command line arguments as delivered in all C and C++ programs. If you have not already done so, this routine initializes MPI with a call to `MPI_Init()` and initializes any logging routines that have been specified (see §6 for more information on logging within *BlockSolve95*).

The final call to a *BlockSolve95* routine must be the routine `BSfinalize()`.

```
int BSfinalize();
```

This routine calls `MPI_Finalize()` if *BlockSolve95* initialized MPI.

## 3.2 The *BlockSolve95* Context `BSprocinfo`

The *BlockSolve95* context data structure, `typedef BSprocinfo`, contains all the information about the parallel environment, options for computing the preconditioner, and settings for the iterative solver. Before calling any *BlockSolve95* routines, the user must first allocate a context of data type `BSprocinfo` for *BlockSolve95* using the routine `BScreate_ctx()`.

```
BSprocinfo  *BScreate_ctx();
```

After the last *BlockSolve95* routine has been called, the context should be freed by calling the routine `BSfree_ctx()`.

```
void  BSfree_ctx(BSprocinfo *context);
```

After the call to `BScreate_ctx()`, you can then call one of many routines to modify the context. We provide default settings for the context that we think will, in general, provide the best performance, but you may benefit from changing some of the settings.

The context options can be broken into three broad categories: (1) the parallel environment, (2) options concerning the computation of preconditioners, and (3) settings for calls to an iterative solver. In the following three subsections we describe the possible options and give the routine names that modify the context.

### 3.2.1 Parallel Environment Options

The following options concern the general parallel environment in which *BlockSolve95* operates. The settings and routines for changing them are as follows:

- **Processor Set**: Definition of the processors that are participating in this call to *BlockSolve95*. If the number of processors participating is equal to the number of processors that are allocated to the user (this is the default case), this value should be set to `MPI_COMM_WORLD`. If, for example, you wish to work on different matrices on different sets of processors at the same time and perhaps later combine the answers, you must set the procset parameter accordingly. The program example `grid2` solves two linear systems simultaneously on a partitioned set of processors. The default setting for this parameter is `MPI_COMM_WORLD`. To reset the value, call the routine `BSctx_set_ps()`.

  ```
  void  BSctx_set_ps(BSprocinfo *context,ProcSet *ps);
  ```

  You may wish to set the processor set to a duplicate communicator (using `MPI_Comm_dup()`) to ensure that there are no message conflicts.

- **Processor id**: The id number of this processor. The default setting is given by the routine `MPI_Comm_rank()`. To reset the value, call the routine `BSctx_set_id()`.

  ```
  void  BSctx_set_id(BSprocinfo *context,int id);
  ```

- **Number of processors**: The number of processors that are calling *BlockSolve95* with a portion of the matrix. The default setting is given by the routine `MPI_Comm_size()`. To reset the value, call the routine `BSctx_set_np()`.

  ```
  void  BSctx_set_np(BSprocinfo *context,int np);
  ```

- **Error checking**: Simple error checking on the user's matrix structure and on some intermediate data structures. The error checking is not very time consuming and is probably a good option to use for the first few runs. The default setting is `FALSE` (i.e., no error checking). To change this value, call the routine `BSctx_set_err()`.

  ```
  void  BSctx_set_err(BSprocinfo *context,int err);
  ```

- **Print information**: Print information about the coloring, reordering, and linear system solution options. If set to `TRUE` this information is printed during execution. The default setting is `FALSE`. To change this value, call the routine `BSctx_set_pr()`.

  ```
  void  BSctx_set_pr(BSprocinfo *context,int pr);
  ```

- **Print logging information**: Print logging information when the routine `BSprint_log()` is called. If set to `TRUE` this information is printed, the default setting is `FALSE`. To change this value, call the routine `BSctx_set_print_log()`.

  ```
  void  BSctx_set_print_log(BSprocinfo *context,int print);
  ```

- **Print current context options**: The current context options. To print information on how the context is currently set, call the function `BSctx_print()`.

  ```
  void  BSctx_print(BSprocinfo *context);
  ```

### 3.2.2 Factorization Options

The following options concern the computation of incomplete factorizations by *BlockSolve95*. Calls to these routines must be made prior to calling `BSmain_perm()`. The settings and routines for changing them are as follows:

- **Maximum clique size**: The maximum number of rows in a single clique. You may wish to limit this value if the cliques become too large and performance is impaired (an unlikely case in most applications and something that requires understanding the algorithms in *BlockSolve95*). The default setting is `INT_MAX`, which is defined in the system include file `limits.h`. To change this value, call the routine `BSctx_set_cs()`.

```
void  BSctx_set_cs(BSprocinfo *context,int cs);
```

- **Maximum identical node size**: The maximum number of rows combined into an identical node. You may wish to limit this value if the i-nodes become too large and performance is impaired (an unlikely case in most applications and something that requires understanding the algorithms in *BlockSolve95*). The default setting is `INT_MAX`. To change this value, call the routine `BSctx_set_is()`.

```
void  BSctx_set_is(BSprocinfo *context,int is);
```

- **Type of local coloring**: The type of local coloring heuristic used. There are two phases in obtaining a coloring of the matrix graph: a global phase in which the Jones/Plassmann algorithm is used and a local phase where either an incident degree ordering (IDO) coloring or a saturated degree ordering (SDO) coloring is used. In general, the SDO colorings are slightly better but take more time to find. The default setting is IDO. To change this value, call the routine `BSctx_set_ct()`.

```
void  BSctx_set_ct(BSprocinfo *context,int ct);
```

- **Retain data structures**: Information retained during the reordering process. The information saved (if the flag is set to `TRUE`) allows a fast reordering if a matrix with the same structure is to be reordered later. The default setting is `FALSE`. To change this value, call the routine `BSctx_set_rt()`.

```
void  BSctx_set_rt(BSprocinfo *context,int rt);
```

- **No clique/i-node reordering**: No attempt to find cliques or i-nodes. This flag should be set to `TRUE` (i.e., no search for i-nodes or cliques will be made) when you know that the i-node or clique sizes will be 1 or very close to 1 (you may wish to experiment with this). The default setting is `FALSE`. To change this value, call the routine `BSctx_set_si()`.

```
void  BSctx_set_si(BSprocinfo *context,int si);
```

- **Scale linear system**: Scaling of the linear system. The default setting is `TRUE`. To change this value, `BSctx_set_scaling()` may be called.

```
void  BSctx_set_scaling(BSprocinfo *context,int scale);
```

  Note that if this function is set, the routine `BSscale_diag()` must be called to do the scaling. We recommend using the matrix diagonal for scaling so that the absolute value of the diagonal of the scaled system is set to one.

### 3.2.3 Solver Options

The following options concern the solution of linear systems by *BlockSolve95*. Calls to these routines must be made prior to calling `BSpar_solve()` or `BSpar_isolve()`. The settings and routines for changing them are as follows:

- **Maximum number of iterations**: The maximum number of iterations allowed the iterative solver. The default setting is 100. To change this value, call the routine `BSctx_set_max_it()`.

```
void   BSctx_set_max_it(BSprocinfo *context,int max_it)
```

- **GMRES restart value**: The maximum number of vectors allowed GMRES before restarting. The default setting is 20. To change this value, call the routine `BSctx_set_restart()`.

```
void   BSctx_set_restart(BSprocinfo *context,int restart);
```

- **Initial guess for iterative method**: The initial vector used by the iterative method. If this flag is `TRUE`, the initial guess is the zero vector. If `FALSE`, the iterative method uses the vector passed to it. This option is useful if a good estimate of the solution is available. The default setting is `TRUE`. To change this value, call the routine `BSctx_set_guess()`.

```
void   BSctx_set_guess(BSprocinfo *context,int guess);
```

- **Convergence tolerance**: The relative residual tolerance requested from the iterative solver. The default setting is `1.0e-5`. To change this value, call the routine `BSctx_set_tol()`.

```
void   BSctx_set_tol(BSprocinfo *context,FLOAT tol);
```

  If the linear system being solved is $Ax = b$, the relative residual is $||Ax - b||/||b||$.

- **Number of RHS vectors**: The number of right-hand side (RHS) vectors to be input to the linear solver. The iterative solvers in *BlockSolve95* can solve for multiple RHS simultaneously. The default setting is 1. This routine should not have to be called by the user because it is set to the correct value if the routine `BSsetup_block()` is called. The program example `grid5` demonstrates how to solve for various numbers of RHS vectors. To change this value, call the routine `BSctx_set_num_rhs()`.

```
void   BSctx_set_num_rhs(BSprocinfo *context,int num_rhs);
```

- **Preconditioner**: The preconditioner to be used by the iterative solver. The choices are incomplete Cholesky (`PRE_ICC`), incomplete LU (`PRE_ILU`), SSOR (`PRE_SSOR`), or block Jacobi (`PRE_JACOBI`). The default setting is `PRE_ICC`. Note that the preconditioner specified must agree with the preconditioner computed. For example, if `PRE_ICC` is specified but an ILU preconditioner is passed to the solver, a *BlockSolve95* error will be returned. To change this value, call the routine `BSctx_set_pre()`.

```
void   BSctx_set_pre(BSprocinfo *context,int pre);
```

- **Iterative method**: The iterative method to be used. The choices are conjugate gradients (`CG`), GMRES (`GMRES`), or SYMMLQ (`SYMMLQ`). The default setting is `CG`. To change this value, call the routine `BSctx_set_method()`.

```
void   BSctx_set_method(BSprocinfo *context,int method);
```

# 4 The User Matrix Data Structure BSspmat

To pass your matrix to *BlockSolve95*, you must use the matrix data structure `typedef BSspmat`. Given this data structure, *BlockSolve95* will convert the matrix to a *BlockSolve95* internal data structure of `typedef BSpar_mat` in the routine `BSmain_perm()` (we discuss this routine in detail in §5). In this section we describe two approaches for converting sparse matrix data into the `BSspmat` format. The first, using the routine `BSeasy_A()`, is recommended and should be fairly painless starting with most standard sparse matrix storage schemes. The second, directly inserting the data and mappings into the structure `BSspmat`, allows for greater flexibility but is more complicated.

## 4.1 The Easy Way to Convert Your Data to BSspmat

We recommend that you use, if possible, the routine `BSeasy_A()` to generate the `BSspmat` data structure. The following are the essential data that you must specify for `BSeasy_A()`.

- The matrix rows must be partitioned onto processors such that each processor has at least one row, each row has a unique global number, and each row is assigned to a unique processor.
- The global row numbers assigned to each processor must be contiguous.[4] If processor $p$ is assigned $n_p$ rows, and the rows numbers start at $i_p$, the rows assigned to processor $p$ must be $i_p, i_p+1, \ldots, i_p+n_p-1$. For example, if there are two processors, processor 0 could have rows $0, 1, 2$ and processor 1 could have rows $4, 5, 6$ but not rows $4, 6, 7$. Note that the global row numbering can skip numbers; in the above example we did not use the row value 3.
- The nonzero structure and matrix values for the rows assigned to each processor must be given in the three arrays `rp`, `cval`, and `aval`. The storage format for these arrays corresponds to the standard compressed row storage (CRS) scheme for sparse matrices. See pages 58–69 of [2] for a detailed discussion of various sparse storage schemes. Each processor uses the *local* numbering of its rows $0, 1, \ldots, n_p - 1$ to index these arrays. The first array `rp` is an index of pointers into the two other arrays indicating where the global column numbers and nonzero values for each row begins. Thus, the $i$th row has nonzeros in columns `cval[rp[i]], cval[rp[i]+1],...,cval[rp[i+1]-1]`. The nonzero values are given in corresponding locations in the array `aval`. The column values for each row in the array `cval` must be sorted from lowest to highest. Note that the entire matrix row must be represented in these arrays even if the matrix is symmetric.

Given this data, the routine `BSeasy_A()` can be called by each processor to generate the `typedef BSspmat` data structure. The function arguments for `BSeasy_A()` are as follows.

```
BSspmat *BSeasy_A(int start_num,int n,int *rp,int *cval,
                  FLOAT *aval,BSprocinfo *procinfo);
```

Note that `start_num` is the starting global row number ($i_p$ in the above example), and `n` is the number of rows assigned the processor ($n_p$ in the above example).

To illustrate the information `BSeasy_A()` requires, we consider the following $4 \times 4$ matrix:

$$A = \begin{pmatrix} a & b & 0 & 0 \\ c & d & e & 0 \\ 0 & f & g & h \\ 0 & 0 & i & j \end{pmatrix} . \tag{1}$$

[4] This limitation applies only to `BSeasy_A()`. *BlockSolve95* itself does not require that row numbers be contiguous; an arbitrary row mapping can be used.

Let the global numbering of the rows and columns of the matrix be 0 to 3 (e.g., $A_{12} = e$). Suppose that we have two processors, 0 and 1, and that we assign rows 0 and 1 of the matrix to processor 0 and rows 2 and 3 to processor 1. Given this distribution of the rows of the matrix and the global numbering of the columns, we would have the arrays **rp**, **cval**, and **aval** as shown in Figure 2. When processor $p$ calls `BSeasy_A()`, its value of **start_num** would be given by $i_p$ and its value of **n** would be $n_p$.

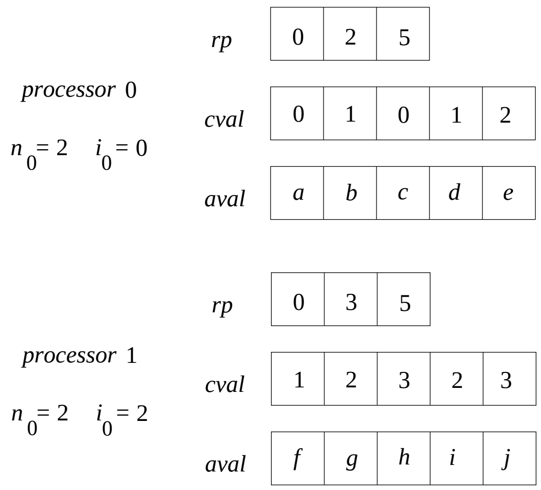


Figure 2: The values of the arrays *rp*, *cval*, and *aval* for the matrix given in equation 1. The matrix is partitioned so that processor 0 is assigned rows 0 and 1 and processor 1 has rows 2 and 3. Note that indexing of the arrays and the matrices rows and columns begins with 0.

## 4.2 The Advanced Way to Convert Matrix Data to `BSspmat`

You can also allocate the `BSspmat` without using `BSeasy_A()`. Although this approach requires a bit more work, it can be more flexible. For example, we had no difficulty in writing a C interface routine to take a matrix written in a standard sequential format by a Fortran code and put this structure around it without duplicating the data in the Fortran sparse matrix.

Within the `BSspmat` data structure, each row of the matrix is represented by the structure `typedef BSsprow`. The data structures are specified as follows:

```
typedef struct __BSsprow {
    int diag_ind;   /* index of diagonal in row */
    int length;     /* num. of nz in row */
    int *col;       /* col numbers */
    double  *nz;    /* nz values */
} BSsprow;

typedef struct __BSspmat {
    int num_rows;       /* number of local rows */
    int global_num_rows;/* number of global rows */
    int symmetric;      /* if TRUE, the matrix should be symmetric */
    int icc_storage;    /* if TRUE, storage scheme used for ICC preconditioner */
                      /* if FALSE, ILU storage scheme used */
    BSmapping   *map;   /* mapping from local to global, etc */
```

```
    BSsprow **rows;     /* the sparse rows */
} BSspmat;
```

First, we discuss the structure `BSspmat`. The field `num_rows` contains the number of rows local to the processor. The field `global_num_rows` contains the total number of rows in the linear system. The fields `symmetric` and `icc_storage` respectively indicate whether the matrix is symmetric and whether an incomplete Cholesky factorization will be computed. The field `map` contains mapping information that will be discussed later. The field `rows` is an array of pointers to local rows of the sparse matrix.

In the structure `BSsprow`, the field `diag_ind` is the index of the diagonal in this row. We require that every row have a diagonal element (the value of this element could be zero). The field `length` contains the number of nonzero values in this row. The field `col` is a pointer to an array of integer values that represent the column number of each nonzero value in the row. These column numbers must be sorted in ascending order. The field `nz` is a pointer to an array of double-precision values that are the nonzero values in the row.

The mapping structure serves three purposes: (1) the mapping of local row number to global row numbers, (2) the mapping of global row numbers to local row numbers, and (3) the mapping of global row number to processor number. We provide routines for you to set up and perform this mapping (details on these routines are given in the "man" pages). You may, however, set up your own mapping and use your own routines through this data structure. The local row numbers on every processor run from `0` to `num_rows - 1`; the global row numbers run from `0` to `global_num_rows - 1`. Each local row has a corresponding global row number.

```
typedef struct __BSmapping {
    void  *vlocal2global;    /* data for mapping local to global */
    void (*flocal2global)(); /* a function for mapping local to global */
    void (*free_l2g)();      /* a function for free'ing the l2g data */
    void  *vglobal2local;    /* data for mapping global to local */
    void (*fglobal2local)(); /* a function mapping global to local */
    void (*free_g2l)();      /* a function for free'ing the g2l data */
    void  *vglobal2proc;     /* data for mapping global to proc */
    void (*fglobal2proc)();  /* a function mapping global to proc */
    void (*free_g2p)();      /* a function for free'ing the g2p data */
} BSmapping;
```

The field `vlocal2global` is a pointer to data that is passed into the local to global mapping function (if you are doing the mapping, you can make this point to whatever you wish). The field `flocal2global` is a pointer to a function for performing the local to global mapping. The field `free_l2g` is a pointer to a function for freeing the data in the field `vlocal2global`. The function for performing the local to global mapping takes five arguments.

```
int        length;     /* the number of row numbers to translate */
int        *req_array; /* the array of local row numbers to translate */
int        *ans_array; /* the array of corresponding global row numbers */
BSprocinfo *procinfo;  /* the processor information context */
BSmapping  *map;       /* the mapping data structure */
```

The next three fields (`vglobal2local`, `fglobal2local`, and `free_g2l`) are exactly the same except the mapping is from global to local row number. The mapping is performed only for rows that are local to the processor; if the mapping is attempted for a nonlocal global row number, a value of `-1` is placed in the `ans_array`. The arguments to the mapping function are as follows.

```
int        length;     /* the number of row numbers to translate */
int        *req_array; /* the array of global row numbers to translate */
int        *ans_array; /* the array of corresponding local row numbers */
BSprocinfo *procinfo;  /* the processor information context */
BSmapping  *map;       /* the mapping data structure */
```

The last three fields (`vglobal2proc`, `fglobal2proc`, and `free_g2p`) are exactly the same except the mapping is from global row number to processor number.[5] The arguments to the mapping function are as follows.

```
int        length;     /* the number of row numbers to translate */
int        *req_array; /* the array of global row numbers to translate */
int        *ans_array; /* the array of corresponding processor numbers */
BSprocinfo *procinfo;  /* the processor information context */
BSmapping  *map;       /* the mapping data structure */
```

[5]This routine will be called by a processor for only those global row numbers that are local to that processor or for those global row numbers that are connected in the sparse matrix to rows that are local to that processor.

# 5 Manipulating and Solving Sparse Systems

This section is divided into two parts. First, we describe how to set up the matrix and preconditioner for parallel solution. Second, we describe how to solve the linear systems after this setup has taken place.

## 5.1 Manipulation and Setup

The data structure that *BlockSolve95* uses to represent sparse matrices is `typedef BSpar_mat`. The routine that converts the user's data structure `BSspmat` to `BSpar_mat` is `BSmain_perm()`. This routine is called as follows.

```
BSpar_mat *BSmain_perm(BSprocinfo *procinfo,BSspmat *A);
```

This routine colors and permutes the sparse matrix to create a new version of the sparse matrix appropriate for parallel computation. A large number of options are available that are set through the context variable `procinfo`, as described in §3. The user's sparse matrix is not changed permanently by this routine, but may be manipulated and restored during execution.

### 5.1.1 Symmetric and Nonsymmetric Matrices

*BlockSolve95* has *two* versions of its internal data structure based on whether (1) the matrix nonzeros are symmetric and an incomplete Cholesky factor is to be computed, or (2) the matrix nonzeros are to be treated as if they are nonsymmetric (even if the matrix values are actually symmetric) and an incomplete ILU factor is to be computed.

The routine `BSset_mat_icc_storage()` must be called *before* `BSmain_perm()` to inform *BlockSolve95* whether internal incomplete Cholesky storage or the incomplete LU storage is to be used. The calling sequence is

```
void BSset_mat_icc_storage(BSspmat *A,int storage);
```

where the incomplete Cholesky storage will be used if `storage` is `TRUE`, and the incomplete LU otherwise. Note that if the matrix is nonsymmetric and yet the incomplete Cholesky storage is specified, incorrect results will be obtained. *BlockSolve95* assumes that the matrix is symmetric if the incomplete Cholesky option is used to minimize the required storage.

The routine `BSset_mat_symmetric()` must be called to inform *BlockSolve95* whether the matrix is symmetric (even if you use the nonsymmetric ILU internal storage format). The arguments to this routine are

```
void  BSset_mat_symmetric(BSspmat *A,int sym);
```

where `sym` is `TRUE` if the matrix is symmetric and `FALSE` if the matrix is nonsymmetric.

In Table 3 we summarize the possible combination of options for the two functions `BSset_mat_icc_storage()` and `BSset_mat_symmetric()`. The tricky choice is whether to use ILU for a symmetric matrix. If the system is very indefinite, you may be forced to this approach to be able to compute an incomplete factor without breakdown. However, remember that this approach requires essentially twice the storage of using the incomplete Cholesky option.

### 5.1.2 Fast Reorderings for Different Matrix Nonzero Values

Often you may wish to solve more than one linear system with the same nonzero structure but different nonzero values. In this case the context should be set to "retain" important intermediate data structures by using the function `BSctx_set_rt()` described in §3. If `BSmain_perm()`

Table 3: Consequences of choosing incomplete Cholesky or incomplete LU for symmetric or nonsymmetric matrices

| BSset_mat_symmetric() | BSset_mat_icc_storage() | Consequences |
|---|---|---|
| TRUE | TRUE | Positive definite preconditioner |
| TRUE | FALSE | ILU computed, twice the storage of incomplete Cholesky |
| FALSE | TRUE | Incorrect results |
| FALSE | FALSE | ILU computed, only correct choice |

has already been called with the "retain" parameter set to true, you can call `BSmain_reperm()` to permute a matrix with the same structure. The routine `BSmain_reperm()` is called with the following arguments.

```
void BSmain_reperm(BSprocinfo *procinfo,BSspmat *A,BSpar_mat *pA);
```

### 5.1.3 Diagonal Scaling of the Linear System

After calling `BSmain_perm()`, the matrix then can be scaled diagonally by calling `BSscale_diag()`. The function arguments are the following.

```
void BSscale_diag(BSpar_mat *A,FLOAT *sc_diag,BSprocinfo *procinfo);
```

Once the matrix has been scaled, *BlockSolve95* automatically solves the scaled system

$$(D^{-1}AD^{-1})(Dx) = (D^{-1}b) \ , \tag{2}$$

where $D$ is the diagonal matrix whose values are the square root of the absolute value of the vector `sc_diag` (usually chosen as the diagonal of $A$). Thus, if the unscaled matrix has a negative diagonal entry, it will be $-1$ after scaling.

### 5.1.4 The Communication Data Structure `BScomm`

Prior to either factoring or solving the matrix, the communication patterns used by *BlockSolve95* must be created. For the factorization phase, this pattern is compiled by calling the routine `BSsetup_factor()`.

```
BScomm *BSsetup_factor(BSpar_mat *A,BSprocinfo *procinfo);
```

For matrix solution, it is compiled by calling `BSsetup_forward()`.

```
BScomm *BSsetup_forward(BSpar_mat *A,BSprocinfo *procinfo);
```

Both routines return a communication pattern of `typedef BScomm`. The communication patterns may be freed by calling `BSfree_comm()`.

```
void BSfree_comm(BScomm *comm_ptr);
```

### 5.1.5 Computing an Incomplete Factorization

If an incomplete factor is to be computed, a copy of the matrix must be made by using the routine `BScopy_par_mat()`.

```
BSpar_mat *BScopy_par_mat(BSpar_mat *A);
```

The copy is necessary because the iterative solver will require both the initial matrix and the factorization. As an aside, we note that (for the incomplete Cholesky storage scheme) the copy of the sparse matrix shares the clique storage space with the matrix from which it is copied. For more information see the `man` page on `BScopy_par_mat()`.

The incomplete factorization is computed by calling the routine `BSfactor()`.

```
int BSfactor(BSpar_mat *A,BScomm *comm,BSprocinfo *procinfo);
```

The matrix `A` will be overwritten with the incomplete factorization. If the factorization is successful the routine returns `0`; otherwise a negative integer is returned whose absolute value is the row number of the color (less one) where the failure occurred. Note that the kind of factorization computed is determined by the internal representation of the matrix produced by `BSmain_perm()`. For example, consider the following code segment.

```
BSset_mat_symmetric(A,TRUE);
BSset_mat_icc_storage(A,FALSE);
pA = BSmain_perm(procinfo,A); CHKERR(0);
/* diagonally scale the matrix */
BSscale_diag(pA,pA->diag,procinfo); CHKERR(0);
/* set up the communication structure for triangular matrix solution */
Acomm = BSsetup_forward(pA,procinfo); CHKERR(0);
/* get a copy of the sparse matrix */
f_pA = BScopy_par_mat(pA); CHKERR(0);
/* set up a communication structure for factorization */
f_comm = BSsetup_factor(f_pA,procinfo); CHKERR(0);
/* compute the incomplete LU factorization */
ierr = BSfactor(f_pA,f_comm,procinfo);
```

In this example the call to `BSset_mat_symmetric()` specifies the matrix as being symmetric. The internal storage format for the incomplete LU factorization is chosen prior to the call to `BSmain_perm()` by calling the routine `BSset_mat_icc_storage()`. Thus, the call to `BSfactor()` will compute the incomplete LU factorization rather than the incomplete Cholesky factorization even though the matrix is specified as symmetric. Also, note that the matrix has been diagonally scaled before the factorization with the call to `BSscale_diag()`.

### 5.1.6 What to Do If the Factorization Fails

An attempt to compute the incomplete Cholesky factorization of a positive definite matrix can fail if a zero or negative diagonal is encountered during the factorization. We note that the incomplete factorization has been shown to exist only if the matrix is diagonally dominant or in several other special cases; in general, there is no guarantee that it will exist. In the case of failure the matrix must be recopied and the factorization retried. We recommend diagonally scaling the matrix and using the following loop to accomplish this task.

```
my_alpha = 1.0;
/* get a copy of the sparse matrix */
```

```
f_pA = BScopy_par_mat(pA);
/* factor the matrix until successful */
while (BSfactor(f_pA,f_comm,procinfo) != 0) {
    /* recopy just the nonzero values */
    BScopy_nz(pA,f_pA);
    /* increment the diagonal shift */
    my_alpha += 0.1;
    BSset_diag(f_pA,my_alpha,procinfo);
}
```

In this code segment, we are shifting the diagonal of the matrix by 0.1 every time the factorization fails. Other strategies are certainly possible and could easily be implemented by the user. The routine `BSset_diag()` is used to change the entire diagonal to `my_alpha`.

```
void BSset_diag(BSpar_mat *A,FLOAT my_alpha,BSprocinfo *procinfo);
```

## 5.2 Solving the Linear System

Once the parallel matrix and the communication structures have been created, we are ready to solve the sparse linear system. One of two routines can be called to do this: (1) `BSpar_solve()`, for either symmetric or nonsymmetric linear systems; and (2) `BSpar_isolve()`, for symmetric indefinite systems, especially those involved in "shift and invert" strategies.

`BSpar_solve()` can be used repeatedly to solve systems of linear equations with one or with multiple right-hand sides. The calling arguments are as follows.

```
int BSpar_solve(BSpar_mat *A, BSpar_mat *fact_A, BScomm *comm_A,
    FLOAT *rhs, FLOAT *x, FLOAT *residual, BSprocinfo *procinfo);
```

The routine returns the number of iterations taken by the solver. If the solver fails (for example, a negative curvature direction is found in a conjugate gradient iteration), the solver returns the negative of the iteration number.

The following code segment presents an example of how `BSpar_solve()` could be called.

```
BSctx_set_method(procinfo,GMRES);
BSctx_set_pre(procinfo,PRE_ILU);
BSctx_set_max_it(procinfo,50);
BSctx_set_restart(procinfo,25);
BSctx_set_guess(procinfo,TRUE);
BSctx_set_tol(procinfo,1.0e-7);
num_iter = BSpar_solve(pA,f_pA,Acomm,rhs,x,residual,procinfo); CHKERR();
```

In this example the GMRES iterative method has been specified, preconditioned by an incomplete LU factorization. The preconditioner specified must agree with the preconditioner that has been computed by `BSfactor()`. The maximum number of iterations allowed the solver has been specified as 50, and the number of storage vector allowed GMRES before restart as 25. In this example `BSctx_set_guess()` has specified that the solver use an initial guess for `x` as the zero vector. Finally, the relative residual tolerance, $||Ax - b||/||b||$, has been set to `1.0e-7`. Additional details on the arguments used can be found in the `man` page.

A second solver, `BSpar_isolve()`, is set up to solve the shifted system $(A - \sigma B)x = b$, where $A$ and $B$ are symmetric matrices, $\sigma$ is a real constant, $x$ is the solution vector, and $b$ is the right-hand side. *BlockSolve95* is set up to take advantage of $B$ being NULL or $\sigma$ being zero. The calling sequence is the following.

```
int BSpar_isolve(BSpar_mat *A, BSpar_mat *fact_A, BScomm *comm_A,
    BSpar_mat *B, BScomm *comm_B, FLOAT *in_rhs, FLOAT *out_x,
    FLOAT shift, FLOAT *residual, BSprocinfo *procinfo);
```

BSpar_isolve() uses the SYMMLQ algorithm, which requires that the preconditioner, if any, be positive definite. Symmetric diagonal scaling is not possible for an indefinite matrix, so one of the other preconditioners must be used. The restriction that the preconditioner be positive definite is too restrictive for many problems, but we know of no general-purpose alternative to SYMMLQ that takes advantage of symmetry while allowing an indefinite preconditioner. Another option would be to explicitly form $(A - \sigma B)$ and solve the resulting system with GMRES preconditioned with ILU (at the expense of twice the memory). The program examples grid7 and grid8 compare these two options. See §8 for details.

If you wish simultaneously to solve for more than one right-hand side, you must call the routine BSsetup_block() to modify the communication structure to accommodate the multiple right-hand sides. The arguments for this function are the following.

```
void BSsetup_block(BSpar_mat *A,BScomm *comm,int block_size,
                   BSprocinfo *procinfo);
```

## 5.3 Freeing Matrices

To free the parallel matrix created by BSmain_perm(), call the routine BSfree_par_mat().

```
void BSfree_par_mat(BSpar_mat *A);
```

To free a copy of a parallel matrix created by BScopy_par_mat(), call the routine BSfree_copy_par_mat().

```
void BSfree_copy_par_mat(BSpar_mat *A);
```

# 6 Error Checking, Flop Counting, Message Passing, and the BLAS

In this section we discuss a number of useful, miscellaneous features available in *BlockSolve95*. These features include error checking, flop counting, matrix size and ordering information, message-passing options, and BLAS options for optimizing processor performance.

## 6.1 Error Checking within *BlockSolve95*

*BlockSolve95* uses an error-checking system based on the two macros `SETERR()` and `CHKERR()`, which are defined in the file `include/BSdepend.h`. When *BlockSolve95* is compiled with the flag `DEBUG_ALL` defined in the file `include/BSdepend.h`, then when an internal error occurs (such as a failed *malloc()* call), *BlockSolve95* returns to the user, and the error code can be checked and a traceback returned. The traceback includes the routine names and line numbers where the error occurs; this information can be useful if you suspect an error in *BlockSolve95*. We highly recommend the use of `DEBUG_ALL` until you are extremely sure of your code; even then, it is inexpensive to use `DEBUG_ALL` with *BlockSolve95*. The default is that `DEBUG_ALL` is set to `TRUE` in `include/BSdepend.h`.

## 6.2 Flop Counting

When the *BlockSolve95* is compiled with `MLOG` defined, flops (floating-point operations) are being logged. (See §7 for more information about compile-time options.) You can access the current number of flops executed by *BlockSolve95* on a particular processor with the call to the function

```
double BSlocal_flops();
```

which returns a double. The total number of flops performed on all processors can be obtained by calling the function

```
double BSglobal_flops(BSprocinfo *procinfo);
```

which also returns a double.

The following code segment demonstrates how you can estimate flop rates for a section of code.

```
init_flops = BSglobal_flops(procinfo);
init_time = MPI_Wtime();
num_iter = BSpar_solve(pA,f_pA,Acomm,rhs,x,residual,procinfo); CHKERR(0);
total_time = MPI_Wtime() - init_time;
total_flops = BSglobal_flops(procinfo) - init_flops;
tmflop = total_flops/(total_time);
```

In this case, `tmflop` would be an estimate of the combined flop rate for all the processors involved in the call to `BSpar_solve()`.

*BlockSolve95* also does some rudimentary logging of timing and flop counts of routines within the package when compiled with `MLOG` defined. By calling the function

```
int BSprint_log(BSprocinfo *procinfo);
```

before the final call to `BSfree_ctx()`, this logging information can be printed out.

## 6.3 Matrix Statistics

*BlockSolve95* includes a number of functions that can be called to obtain information about a matrix of `typedef BSpar_mat`. For example, the following two functions may be called to determine the matrix nonzeros locally assigned to a processor and the total (global) number of nonzeros in the matrix.

```
int BSlocal_nnz(BSpar_mat *);
int BSglobal_nnz(BSpar_mat *,BSprocinfo *);
```

The following two functions may be called to obtain the number of i-nodes locally assigned to a processor and the global number of i-nodes.

```
int BSlocal_num_inodes(BSpar_mat *);
int BSglobal_num_inodes(BSpar_mat *);
```

The following two functions return the the number of cliques locally assigned to a processor and the global number of cliques.

```
int BSlocal_num_cliques(BSpar_mat *);
int BSglobal_num_cliques(BSpar_mat *);
```

The following function returns the number of colors used to color the graph associated with matrix clique structure.

```
int BSnum_colors(BSpar_mat *);
```

## 6.4 Blocking and Nonblocking Messages

*BlockSolve95* can be compiled so that it does not use blocking (synchronous) sends and receives. (In MPI the blocking send is the function `MPI_Send()`; the nonblocking (asynchronous) send is the function `MPI_Isend()`.) The MPI standard does not require that two processors simultaneously issuing blocking sends buffer messages and return; therefore deadlock can result. This problem can be avoided by using nonblocking sends, at the expense of more memory (because the message buffers cannot be freed until it is certain that the messages have been received). The IBM SP requires the use of such nonblocking sends.

*BlockSolve95* compiles with the blocking or nonblocking send versions based on whether `NO_BLOCKING_SEND` is defined in the include file `include/BSdepend.h`. If you are concerned about performance or memory usage, you should experiment with the difference between blocking and nonblocking sends on your particular architecture.

## 6.5 MPI Communicators and Message Number Conflicts

*BlockSolve95* currently does not make a copy of the user's MPI communicator; instead it uses as a default the current `MPI_COMM_WORLD`. If there is any possibility of a message-passing conflict (for example, unsatisfied wild card receives from the user's code), the communicator can be copied by using `MPI_Comm_dup()` and can be handed to *BlockSolve95* by using `BSctx_set_ps()`.

*BlockSolve95* uses message numbers beginning at 10,000. It uses a significant but variable number of messages after that. Currently the number of messages used is 20+(10000*number_of_processors). The number of messages needed by *BlockSolve95* depends on the problem being solved, but if the number of messages allocated to it is too small, it will detect an error and return accordingly (if `DEBUG_ALL` is on). The current setting of 10,000 is generous. The message numbers as well as the number of messages can be changed simply by altering `include/BSprivate.h`. The avoidance of conflicts in the use of message numbers can be ensured by copying the communicator as described above.

## 6.6 Inline Macros for the BLAS

The performance of the vendor-supplied BLAS for small systems can be disappointing because of the subroutine call overhead, error checking, and special case handling (overloading) in the implementation of these BLAS. In addition, *BlockSolve95* has to gather and scatter noncontiguous vector data (although not matrix data) to use the dense BLAS. A feature in *BlockSolve95* is special handling of the Level-2 BLAS for small i-node sizes; *BlockSolve95* uses macros to put code inline for the special BLAS cases required. The performance improvement can be substantial. For example, in Table 4 we show the processor performance improvement obtained on the Argonne IBM SP system (RS/6000-370 processor).

Table 4: Comparison of the single-processor performance of *BlockSolve95* conjugate gradient and GMRES iterations, with vendor-supplied, Level-2 BLAS (GEMV and TRMV) and the new inline macros. The symmetric systems are solved with the conjugate gradient method; the nonsymmetric systems are solved with GMRES.

| Size of i-node | Symmetric (Y or N) | Vendor BLAS (Mflops) | BS95 Macros (Mflops) | Improvement Ratio |
|---|---|---|---|---|
| 2 | Y | 3.5 | 8.3 | 2.35 |
| 3 | Y | 6.1 | 12.5 | 2.06 |
| 5 | Y | 10.3 | 18.3 | 1.77 |
| 6 | Y | 14.7 | 22.3 | 1.52 |
| 7 | Y | 17.6 | 23.2 | 1.32 |
| 2 | N | 5.2 | 7.1 | 1.37 |
| 3 | N | 8.0 | 10.6 | 1.33 |
| 5 | N | 16.1 | 19.1 | 1.19 |

These macros are defined in the `include/BSmy_blas.h`. The default is that libraries are compiled with the inline versions for small i-node sizes (less than 10); otherwise, the vendor-supplied BLAS are used. These macros can be turned off by editing this file and not defining `MY_BLAS_DTRMV_ON` and `MY_BLAS_DGEMV_ON`. Also, the maximum level of unrolling can be changed to tune performance for a particular architecture.

# 7 Installation

Underneath the main `BlockSolve95` directory are six other directories: (1) `src`, which contains the source code and makefiles for *BlockSolve95*, (2) `doc`, which contains the documentation for *BlockSolve95*, (3) `examples`, which contains example programs that demonstrate the use of *BlockSolve95*, (4) `include`, (5) `lib`, and (6) `bmake`.

*BlockSolve95* uses the same makefile system as PETSc [1]. The `README` file in the main directory gives details on how to build the libraries. The makefiles will build the *BlockSolve95* library in a subdirectory of the `lib` directory based on the machine architecture and the level of optimization. The machine architecture should be specified by the environmental variable `$PETSC_ARCH`. You will have to modify the file `bmake/$PETSC_ARCH/$PETSC_ARCH.site` as directed in the `README` file.

Several compiler options affect *BlockSolve95*. The `DEBUG_ALL` flags were described in §6. The flag `MLOG` is associated with the logging facilities within *BlockSolve95*, and more information can be found on them in the file `include/BSlog.h`. A preprocessor variable called `BSDOUBLE` is defined in `include/BSsparse.h`. If `BSDOUBLE` is defined, *BlockSolve95* will compile a double-precision version; otherwise, a single-precision version is compiled. Unfortunately, the routine names for both versions are the same.

There is special case on the Cray T3D because the C code requires that `FLOAT` be defined as `double` but the BLAS and LAPACK routines require the single-precision names (although they are the double-precision versions). Details can be found in the include file `include/BSsparse.h`.

## 7.1 Other Libraries

To run *BlockSolve95*, you need LAPACK; the BLAS 1, 2, and 3 libraries; and an MPI implementation for the particular architecture being used. If MPI has not been installed on your system, we recommend your using MPICH. This package is available via anonymous ftp from `info.mcs.anl.gov` in the directory `pub/mpi`. More information on MPI can be found on the WWW at `http://www.mcs.anl.gov/mpi/index.html`.

## 7.2 UNIX `man` Pages

UNIX `man` pages have been generated for the *BlockSolve95* routines and are located in the directory `doc/man`. To use these pages you must include the path to the directory `BlockSolve95/doc/man` in your `MANPATH` environmental variable. This variable can be set with the command `setenv`. The tool `xman` provides a nice interface to these man pages. The *BlockSolve95* routines can be accessed in Section 3 and the include files in Section h from the pull-down menu given by the "Sections" button on the `xman` toolbar.

## 7.3 Availability of *BlockSolve95*

The *BlockSolve95* package can obtained via anonymous ftp from `info.mcs.anl.gov`. The package is in the directory `pub/BlockSolve95`. In addition, *BlockSolve95* is available from netlib. The current version number and last date of modification are in the file `include/BSsparse.h`. Please send any questions via e-mail to `jones@cs.utk.edu` or `plassman@mcs.anl.gov`. Include your name, affiliation, U.S.-mail address, and e-mail address, along with a description of what you might be interested in doing with *BlockSolve95*.

Information on the current status of *BlockSolve95* can be obtained on the WWW at `http://www.mcs.anl.gov/blocksolve95/index.html`.

### 7.4 Differences between Previous *BlockSolve* Versions

*BlockSolve95* incorporates many improvements over *BlockSolve* v1.1 and v2.0. Among these improvements are the following.

- A compile-time option has been added that directs *BlockSolve95* to use no blocking sends (see §6 or the include file `include/BSdepend.h` for details). This feature is necessary because certain machines (e.g., the IBM SP series architecture) have inadequate message buffering space allocated to use blocking sends.
- *BlockSolve95* now uses the MPI message-passing standard instead of *Chameleon* for communication.
- *BlockSolve95* can now compute incomplete LU factorizations for matrices that have a symmetric nonzero pattern but nonsymmetric entries. Also, the iterative solver GMRES has been added, which can be used to solve these nonsymmetric systems. See §5 for details.
- A subroutine, `BSeasy_A()`, has been added that allows the user to convert a sparse matrix stored in a familiar storage format to the *BlockSolve95* data structure used for parallel computation (see §4 for details). In addition, the routine `BSfree_easymat()` has been added to free the data structures that *BlockSolve95* allocates when `BSeasy_A()` is called.
- The matrix reordering routines called in `BSmain_perm()` have been significantly modified to greatly reduce the runtime of *BlockSolve95* for problems with only one degree of freedom per grid point. *BlockSolve95* is still designed for more complex problems, but the reordering time for these simpler problems is now more satisfactory.
- The code is now ANSI-C. This should result in fewer user errors because of the type checking that can now be done on argument lists. Note that the user's code does not have to be ANSI-C to use the libraries.
- *BlockSolve95* has special macros for handling the level-2 BLAS GEMV and TRMV for small i-node sizes. The performance of the vendor-supplied BLAS have proven to be inadequate in this parameter range, so inline versions of specific BLAS are now included in the include file `include/BSmy_blas.h`.
- The set of example programs is greatly expanded and can be used to test most aspects of *BlockSolve95*. See §8 for a description of the program examples.

# 8 Example Programs

*BlockSolve95* includes a number of example programs that demonstrate most aspects of the software library. The code for these examples is contained in the **examples** directory.

```
A description of BlockSolve95 example programs, a description of
the problem calling arguments is given below.
=================================================================

grid0 -- Simple example.  Uses 3-d 7-point finite-difference
  stencil.  BlockSolve context set not to look for cliques or
  inodes.  Symmetric problem.

  . mpirun -np P grid0.ARCH PX PY PZ NX NY NZ

-----------------------------------------------------------------
grid1 -- An unbalanced grid example.  The processors on the left
  and right ends of the grid use a 7 pt stencil, those in the middle
  use a 27-pt stencil.  BlockSolve context set not to look for cliques
  and inodes.  Symmetric problem.

  . mpirun -np P grid1.ARCH PX PY PZ NX NY NZ

-----------------------------------------------------------------
grid2 -- A demonstration of running BlockSolve on two independent
  sets of processors simultaneously.  This problem is partitioned
  in only the x and y directions.  The processors on the left
  and right ends of the grid use a 7 pt stencil; those in the middle
  use a 27-pt stencil.  BlockSolve context set not to look for
  cliques or inodes.  Symmetric problem.

  . mpirun -np P grid2.ARCH PX PY NX NY NZ

-----------------------------------------------------------------
grid3 -- A 2-D grid distributed across the processors.  The 2-D
  grid is partitioned in both dimensions among the processors.
  The number of processors *must* be the square of an integer,
  e.g., 9 but not 8.  Either a 5-pt or 9=pt stencil can be
  specified.  BlockSolve context set not to look for cliques or
  inodes.  Symmetric problem.

  . mpirun -np P grid3.ARCH NXY NST

-----------------------------------------------------------------
grid4 -- Simple example (same as grid0) but this code uses lower level
  BlockSolve functions to set up the parallel sparse matrix data
  structures (done in get_mat4.c).  Uses 3-d 7-point finite-difference
  stencil.  BlockSolve context set not to look for cliques or inodes.
  Symmetric problem.

  . mpirun -np P grid4.ARCH PX PY PZ NX NY NZ

-----------------------------------------------------------------
grid5 -- Can be used to test a variety of BlockSolve features.
  The grid problem is based on the 7-pt stencil; however, one
  can specify a number of components at each grid points.  Each
```

```
  of these components will have an identical nonzero structure
  (inode) in the sparse matrix.  One can specify either a symmetric
  or nonsymmetric nonzero structure.  One can specify whether the
  BlockSolve is to look for inode/cliques or not.

 . mpirun -np P grid5.ARCH PX PY PZ NX NY NZ SYM NC IN PRE METHOD SCALE NUM_RHS

-----------------------------------------------------------------
grid6 -- A demonstration of setting up the BlockSolve communication
  structure and using the same structures to solve multiple linear
  systems with the same nonzero structure but different nonzero
  values.  In this case, we compute a sequence of preconditioners
  by varying the diagonal shift applied to the matrix used to
  compute in the computation of the incomplete factor.

 . mpirun -np P grid6.ARCH PX PY PZ NX NY NZ SYM IN NC
-----------------------------------------------------------------
grid7 -- This example uses SYMMLQ to solve the symmetric shifted
  system (A-s*B)x = b, where s is a shift and A and B are symmetric.
  (Note that this system is indefinite for large enough shift s.)
  Systems of this kind form the core computation in a shift
  and invert strategy within a Lanzcos iteration.  In the usual
  case, A would be the assembled stiffness matrix and B would be
  the associated mass matrix.  For this example, B has the same
  nonzero structure of A and its values are the absolute values
  of the values of A.  Note that SYMMLQ requires that the preconditioner
  (the incomplete factorization of A) be positive definite!

 . mpirun -np P grid7.ARCH PX PY PZ NX NY NZ PRE IN NC BMAT SHIFT
-----------------------------------------------------------------
grid8 --  This example solves the same problem as that in grid7
  except here we explicitly form the indefinite matrix C=(A-s*B) and
  solve the resulting linear system system, Cx = b, using GMRES.
  One can use either ILU or ICC as a preconditioner, although
  for large shifts the diagonal of C must be shifted to avoid
  breakdown in the computation of the incomplete Cholesky factor.

 . mpirun -np P grid8.ARCH PX PY PZ NX NY NZ PRE IN NC SHIFT
-----------------------------------------------------------------

A description of possible example program arguments:

  .  ARCH = Machine architecture, e.g., sun4, rs6000, IRIX, etc.
  .  P    = number of processors
  .  PX   = the number of processors in the x direction
  .  PY   = the number of processors in the y direction
  .  PZ   = the number of processors in the z direction
  .  NX   = number of points in the x direction on each processor
  .  NY   = number of points in the y direction on each processor
  .  NZ   = number of points in the z direction on each processor
  .  NXY  = number of points in the x and y directions on each processor
  .  NST  = number of points used in the stencil
  .  SYM  = 0, use symmetric data structure, = 1, use nonsymmetric
  .  NC   = number of components per grid point
  .  IN   = 0, do not use, = 1, use inode/clique stuff
```

```
.  PRE  = 0, use ICC, = 1, use ILU
.  METHOD = 0, use CG, use GMRES
.  SCALE = 0, do not scale system, = 1, scale system
.  NUM_RHS = number of RHSs to solve for simultaneously
.  BMAT = 0 = use identity for shift, 1 = use positive of A
.  SHIFT = shift to use in the shift and invert system
```

# 9 Limitations and Future Plans

The user should be aware of a few limitations in *BlockSolve95*:

- Each row of the matrix must have a diagonal entry. That entry may be zero, but it must be explicitly represented in the matrix structure.
- If the matrix is indefinite, one cannot solve for a block of vectors simultaneously in the current code.
- *BlockSolve95* does not check for or catch exceptions associated with floating-point errors.
- Each processor involved in a BlockSolve call must have at least one row of the matrix.
- *BlockSolve95* does not allow the user to mix machines with different byte orderings or different sizes for datatypes in the same computation.

Another limitation involves coloring options. It is possible with the current version that if the portion of the matrix structure contained on some processors is very different from the structure contained on other processors, then the number of colors on each of these processors can be quite different. Such a situation could arise if different-order finite elements are used on different processors (but would not arise just by applying boundary conditions to some processors, but not to others). This imbalance in the number of processors could degrade performance. A balanced coloring heuristic that addresses this situation is described in [3]; however, this heuristic is not yet included in *BlockSolve95*.

## 10 Related Software

A number of software packages are closely related to *BlockSolve95* and may be of use in particular applications.

- The software package PETSc [1] contains an interface to *BlockSolve95*. It can be used to access a larger number of iterative solvers, nonlinear methods, and a large number of other useful features. Information on PETSc may be obtained on the WWW at `http://www.mcs.anl.gov/petsc/petsc.html`.
- The SUMAA3d (Scalable Unstructured Mesh Algorithms and Applications) project is developing algorithms and software for many of the tasks associated with unstructured mesh computation. Current information about this project can be obtained on the WWW at `http://www.mcs.anl.gov/sumaa3d/index.html`.
- Preliminary adaptive refinement software has been developed for finite element methods in two and three dimensions [12]. This software is part of the SUMAA3d project.

## Acknowledgments

We acknowledge Wing-Lok Wan for the monumental summer project of adapting the incomplete Cholesky factorization to the incomplete LU case. We thank William Gropp for his help with the MPI message-passing implementation and for his inspiration on a large number of software issues. We thank Lois Curfman McInnes, Barry Smith, and the PETSc team for the makefile facilities, motivation for conversion to ANSI-C, help with the documentation and testing, and numerous suggestions for improving the software. We also thank Satish Balay for discussions on the implementations of the required BLAS macros.

# References


[1] S. Balay, L. Curfman McInnes, W. D. Gropp, and B. F. Smith, *PETSc 2.0 users manual*, ANL Report ANL-95/11, Argonne National Laboratory, Argonne, Ill., Nov. 1995.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[3] R. K. Gjertsen, Jr., M. T. Jones, and P. E. Plassmann, *Parallel heuristics for improved, balanced graph colorings*, Journal of Parallel and Distributed Computing, 37 (1996), pp. 171–186.

[4] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1994.

[5] B. Hendrickson and R. Leland, *The Chaco user's guide: Version 2.0*, Sandia Report SAND94–2692, Sandia National Laboratories, Albuquerque, N.M., June 1995.

[6] M. T. Jones and P. E. Plassmann, *The effect of many-color orderings on the convergence of iterative methods*, in Proceedings of the Copper Mountain Conference on Iterative Methods, SIAM LA-SIG, 1992.

[7] ——, *Solution of large, sparse systems of linear equations in massively parallel applications*, in Proceedings of Supercomputing '92, IEEE Computer Society Press, 1992, pp. 551–560.

[8] ——, *Computation of equilibrium vortex structures for type-II superconductors*, The International Journal of Supercomputer Applications, 7 (1993), pp. 129–143.

[9] ——, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[10] ——, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

[11] ——, *Scalable iterative solution of sparse linear systems*, Parallel Computing, 20 (1994), pp. 753–773.

[12] ——, *Parallel algorithms for adaptive mesh refinement*, SIAM Journal on Scientific Computing, 18 (1997), pp. 686–708.

[13] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputing Applications, 8 (1994).

# Subject Index

# Function Index