ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

BlockSolve v1.1: Scalable Library Software for the Parallel Solution of Sparse Linear Systems

MARK T. JONES AND PAUL E. PLASSMANN

Mathematics and Computer Science Division

Report ANL-92/46
December 1992

ABSTRACT

BlockSolve is a software library for solving large, sparse systems of linear equations on massively parallel computers. The matrices must be symmetric, but may have an arbitrary sparsity structure. BlockSolve is a portable package that is compatible with several different message-passing pardigms. This report gives detailed instructions on the use of BlockSolve in applications programs.

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

1 Introduction

BlockSolve is a scalable parallel software library for the solution of large sparse, symmetric systems of linear equations. It runs on a variety of parallel architectures and can easily be ported to others. *BlockSolve* utilizes the *Chameleon* package [3] to achieve portability across architectures and compatibility with message-passing paradigms such as `p4` [1] and PVM [8], as well as the message-passing primitives available on architectures such as the Intel iPSC/860. A user does not need to use the *Chameleon* package to use *BlockSolve*; all that is required is that *BlockSolve* be compiled with the correct options to make it compatible with the message-passing paradigm and architecture that it will be used on.

BlockSolve is primarily intended for the solution of sparse linear systems that arise from physical problems having 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 will typically have anywhere from 3-6 degrees of freedom associated with it. *BlockSolve* is written to take advantage of problems of this nature; however, it can be reasonably efficient for problems that have only one degree of freedom associated with each node, such as the three-dimensional Poisson problem. We do not require that the matrices have any particular structure other than being sparse and symmetric.

BlockSolve is intended to be used within real application codes. It has been our experience that most application codes need to 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. We have therefore designed *BlockSolve* to work best within this context.

In the next section we will give a brief description of the algorithms in *BlockSolve*, as well as references to more information. In §3 we describe how to use *BlockSolve* and give descriptions of the necessary data structures. Information relevant to the installation and testing of *BlockSolve* is given in §4. In §5 we list some of the limitations of *BlockSolve* and detail our future plans.

2 Algorithm Descriptions

BlockSolve utilizes the preconditioned conjugate gradient algorithm for symmetric positive definite matrices and the preconditioned SYMMLQ algorithm for symmetric indefinite matrices. For basic information on these algorithms, we refer the reader to [2]. One important note is that the SYMMLQ algorithm requires a positive definite preconditioner, and this requirement can be a serious limitation if the matrix being solved is very indefinite.[1]

The user has the option of selecting a combination of four preconditioners. The

[1] By "very indefinite," we mean that the matrix has many negative and many positive eigenvalues.

first option is a simple diagonal scaling of the matrix. We advocate always diagonally scaling the matrix, whether or not one of the others preconditioners is selected. The other preconditioning options are incomplete Cholesky factorization, SSOR ($\omega = 1$), and block Jacobi (where the blocks are the cliques of the graph associated with the sparse matrix). We recommend that the user select the incomplete Cholesky factorization with diagonal scaling for symmetric positive definite matrices.[2] This is the algorithm that *BlockSolve* was designed for, and it has proven useful for a variety of practical problems.

BlockSolve does not partition the matrices across the processors for the user. *BlockSolve* simply accepts an already partitioned matrix with the assumption that the partitioning is a good one; its performance is limited by the quality of the partition. We believe that this is a reasonable approach for the linear system solver because the user must also have a good partition for the other aspects of an application to perform well. Therefore, we view the partitioning problem as a separate, but important problem. We assume that the right-hand side and the solution vector are partitioned in the same manner as the rows of the sparse matrix.

We achieve parallelism in the conjugate gradient (SYMMLQ) portion of the code by partitioning the vectors used in the algorithms in the same manner that the rows of the matrix are partitioned across the processors. Then it is a simple matter of executing inner products and daxpy's in parallel.

To achieve scalable parallel performance in the incomplete Cholesky and SSOR preconditioners, we color the graph of the sparse matrix using a parallel coloring algorithm [7]. The combination of coloring a general symmetric sparse matrix and the incomplete Cholesky algorithm has proven very successful for solving large problems on scalable parallel architectures [4], [6]. We have addressed the issue of convergence of this combination of algorithms in [5].

To achieve good performance on each node, we reorder the matrix to allow the use of the higher-level dense BLAS. This is particularly important 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 identical nodes and cliques in the graph associated with the matrix. Identical nodes typically exist when multiple degrees of freedom are associated with each node in the graph. Cliques are found in many graphs associated with sparse matrices, but larger ones are typically found in graphs where multiple degrees of freedom are associated with each node 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, 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.

[2]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.

3 Using *BlockSolve*

We will first discuss the context data structure that must be created prior to any calls to a *BlockSolve* routine. We will then describe the data structures that contain the user's sparse matrix. These data structures must exist on every process that will be calling *BlockSolve*. Finally, we will discuss the various *BlockSolve* subroutines that can be called to manipulate and solve sparse linear systems. All subroutine and data structure names in *BlockSolve* are prefixed by either "BS" or "BM." Included with the *BlockSolve* software are examples that demonstrate the use of *BlockSolve*.

3.1 The Context

The context structure is used to convey information about the parallel environment as well as option settings for *BlockSolve*. Before calling any *BlockSolve* routines, the user must first allocate a context (a structure called *BSprocinfo*) for *BlockSolve* using the routine *BScreate_ctx()*; it takes no arguments. When the last *BlockSolve* routine is called, the context can be freed by calling *BSfree_ctx()* with the context as the only argument. After calling *BScreate_ctx()*, the user can then call one of several routines to modify the context. We provide default settings for the context that we think will, in general, provide the best performance, but the user may benefit from changing some of the settings. The settings and routines for changing them are as follows:

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

- Number of processors: The number of processors that are calling *BlockSolve* with a portion of the matrix. The default setting is given by the routine NUMNODES from the *Chameleon* package. To reset the value, call the routine *BSctx_set_np()*.

- Processor Set: Definition of the processors that are participating in this call to *BlockSolve*. If the number of processors participating is equal to the number of processors that are allocated to the user (this is the usual case), then this value should be set to NULL. If, for example, the user wishes to work on different matrices on different sets of processors at the same time and perhaps later combine the answers, then the procset parameter must be set accordingly. For more information on procset and its uses, see the *Chameleon* manual. The default setting for this parameter is NULL. To reset the value, call the routine *BSctx_set_ps()*.

- Maximum clique size: The maximum number of rows in a single clique. The user 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 *BlockSolve*). The default setting is INT_MAX. To change this value, call the routine *BSctx_set_cs()*.

- Maximum identical node size: The maximum number of rows combined into an identical node. The user 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 *BlockSolve*). The default setting is INT_MAX. To change this value, call the routine *BSctx_set_is()*.

- Type of local coloring: In the coloring algorithm, there are two phases: 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()*.

- Error checking: If this flag is true, then some simple error checking on the user's matrix structure and some intermediate data structures is done. The error checking is not very time consuming and is probably a good idea to use for the first few runs. The default setting is false. To change this value, call the routine *BSctx_set_err()*.

- Retain data structures: If this flag is true, then information is saved during the reordering process to allow 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()*.

- Print information: If this flag is true, then information about the coloring and reordering is printed during execution. The default setting is false. To change this value, call the routine *BSctx_set_pr()*.

- No clique/inode reordering: If this flag is true, then no attempt is made to find cliques or i-nodes. This flag should be set to true when the user knows that the i-node or cliques sizes will be 1 or very close to 1 (the user may wish to experiment with this). The default setting is false. To change this value, call the routine *BSctx_set_si()*.

3.2 The User Matrix Data Structure

The user's matrix is passed to *BlockSolve* in the following format. The matrix data structure is represented in the structure *BSspmat* and each row of the matrix is represented by the structure *BSsprow*. We believe that this format is flexible enough to be used in a variety of contexts. 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.

```
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 */
    BSmapping   *map;   /* mapping from local to global, etc */
    BSsprow **rows;     /* the sparse rows */
} BSspmat;
```

First, we address 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 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 the user to set up and perform this mapping (details on these routines are given in the "man" pages). The user is free, however, to setup his own mapping and use his 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 the user is doing the mapping, he is free to make this point to whatever he wishes). 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 5 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, then a value of -1 is placed in the *ans_array*. The arguments to the mapping function are

```
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.[3] The arguments to the mapping function are:

```
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 */
```

[3]It is important to note that 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.

3.3 Manipulating and Solving Matrices

This subsection 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 the setup has taken place.

3.3.1 Manipulation and Setup

The first routine that should be called is *BSmain_perm()*, which takes the context and the user's sparse matrix as arguments. This routine colors and permutes the sparse matrix to create a new version of the sparse matrix appropriate for parallel computation. The user's sparse matrix is not permanently changed during this routine, but may be manipulated and restored during execution. If *BSmain_perm()* has already been called with the "retain" parameter set to true, then the user can call *BSmain_reperm()* to permute a matrix with the same structure as was permuted in the original call to *BSmain_perm()*.

After calling *BSmain_perm()*, the matrix can then be diagonally scaled by calling *BSscale_diag()*.

Prior to either factoring or solving the matrix, the communication patterns used by *BlockSolve* must be created. For factorization this can be done by calling *BSsetup_factor()*. For matrix solution, this is done by calling *BSsetup_forward()*. Both routines return the communication pattern. The communication patterns may be freed by calling *BSfree_comm()*.

If an incomplete factor is to be created, then a copy of the matrix must be made. In addition, if the factorization fails as a result of a zero or negative diagonal being encountered during the factorization, the matrix must be recopied and the factorization retried. The following loop accomplishes this task. It is important to note that the copy of the sparse matrix shares the clique storage space with the matrix that it is copied from (for more information see the "man" page on *BScopy_par_mat()*). The routine *BSset_diag()* is used to change the entire diagonal to alpha; in other words, 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.

```
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 */
    alpha += 0.1;
    BSset_diag(f_pA,alpha,procinfo);
}
```

To free the parallel matrix created by *BSmain_perm()*, call the routine *BSfree_par_mat()*. To free a copy of a parallel matrix created by *BScopy_par_mat()*, call the routine *BS-free_copy_par_mat()*.

3.3.2 Solving the Linear System

Once the parallel matrix and the communication structures have been created, it is possible to solve the sparse linear system. One of two routines can be called to do this: (1) *BSpar_solve()* for symmetric positive definite matrices, and (2) *BSpar_isolve()* for symmetric indefinite matrices.

BSpar_solve() can be used repeatedly to solve systems of linear equations with one or with multiple right-hand sides. Details on the arguments used can be found in the "man" page.

BSpar_isolve() is actually set up to solve the system $(A - \sigma B)x = b$, where A and B are symmetric matrices, σ is a real constant, x is the solution value, and b is the right-hand side. *BlockSolve* is setup to take advantage of B being NULL or σ being zero. *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.

If the user wishes to solve with more than one right-hand side simultaneously, then the routine *BSsetup_block()* must be called to modify the communication structure to accommodate the multiple right-hand sides.

3.4 Error Checking within *BlockSolve*

BlockSolve uses the error-checking system defined in the *Chameleon* package. If *BlockSolve* is compiled with the flag DEBUG_ALL defined, then if an internal error occurs (such as a failed *malloc()* call), *BlockSolve* returns to the user and the error code can be checked with the macros available in *Chameleon* (see the *Chameleon* manual for more information on the error checking system). If *BlockSolve* is compiled with DEBUG_TRACEBACK in addition to DEBUG_ALL, then error messages are printed by the routines that encounter the errors, along with routine names and line numbers where the error occurs. This information can be useful if the user suspects an error in *BlockSolve*. We highly recommend the use of DEBUG_ALL and DEBUG_TRACEBACK until one is extremely sure of one's the code, and even then it is inexpensive to use DEBUG_ALL with *BlockSolve*.

3.5 Message Number Conflicts

BlockSolve 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 *BlockSolve* depends on the problem being solved, but if the number of messages allocated to it is too small, then it will detect an error and return accordingly (if DEBUG_ALL is on). The current setting of 10,000 is very generous. The message numbers as well as the number of messages can be changed by altering BSprivate.h. This modification is a very simple task that any user can do. We hope to make the assignment of message numbers and avoidance of conflicts in the use of message numbers automatic as soon as this facility is provided by the *Chameleon* package.

4 Installation and Testing

Underneath the main *block_solve* directory are three other directories: (1) *src*, which contains the source code and makefiles for *BlockSolve*, (2) *doc*, which contains the documentation for *BlockSolve*, and (3) *examples*, which contains example programs that demonstrate the use of *BlockSolve*.

To make the *BlockSolve* library, one should examine the files make* and Makefile in the *src* directory. These Makefiles are well documented. It is likely that the user will have to modify them as directed in the Makefiles themselves. It is necessary to have the *Chameleon* package installed before trying to make *BlockSolve*.

Several compiler options have an effect on *BlockSolve*. The DEBUG flags were described in §3. The flags MLOG, MCOUNT, and MAINLOG are associated with the logging facilities within *BlockSolve*, and more information can be found on them in the file BSlog.h. There are many compiler flags defined by the *Chameleon* package that have an effect on *BlockSolve*; for information on these flags see the *Chameleon* documentation. A preprocessor variable called DOUBLE is defined in BSsparse.h. If DOUBLE is defined, then BlockSolve will compile a double precision version; otherwise, a single-precision version is compiled. Unfortunately, the routine names for both versions are the same.

All the routines in the *Chameleon* package have been tested extensively with thousands of runs within a few applications at Argonne as well as inside the example programs. We believe that the code is error-free at this point, but it is still possible that when *BlockSolve* is used in new applications, previously undiscovered errors may be found. At the time of this writing, we have run the code on the Intel DELTA, the Intel iPSC/860, and a network of Sun Sparcstations. In the case of the Intel machines we instructed the *Chameleon* package to use the Intel message-passing primitives directly wherever possible. On the Sparcstations we instructed the *Chameleon* package to use the `p4` message-passing system to handle the communication. In the near future we would like to test the code on the CM-5 as well as on Sparcstations using the PVM message-passing system.

4.1 Other Libraries

To run *BlockSolve*, one needs the well-known LAPACK and BLAS 1, 2, and 3 libraries as well as the *Chameleon* package written by William Gropp (gropp@mcs.anl.gov). The *Chameleon* package is available via anonymous ftp from info.mcs.anl.gov in the directory pub/mpi.

4.2 Availability of *BlockSolve*

The *BlockSolve* package can obtained from the ftp server info.mcs.anl.gov using an anonymous login. The package is in the directory pub/BlockSolve. The current version number and last date of modification is in the file BSsparse.h. Please send any questions via e-mail to mjones@mcs.anl.gov. Please include your name, affiliation, U.S.-mail address, and e-mail address along with a description of what (if anything) you might be interested in doing with *BlockSolve*.

5 Limitations and Future Plans

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

- 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.
- *BlockSolve* does not check for or catch exceptions associated with floating-point errors.

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. We are currently working on a new coloring algorithm that will address this situation.

We will also be further integrating *BlockSolve* into the *Chameleon* package. This integration should be largely transparent to the user, but will result in less code in the *BlockSolve* package.

A long-term plan is the extension of *BlockSolve* to nonsymmetric systems.

Acknowledgment

We thank William Gropp for his help in using the Chameleon package.

References

[1] R. Butler and E. Lusk. Private communication, 1991.

[2] G. H. Golub and C. F. V. Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1983.

[3] W. Gropp. Private communication, 1992.

[4] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear systems*, Preprint MCS-P277-1191, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991.

[5] ——, *The effect of many-color orderings on the convergence of iterative methods*, in Proceedings of the Copper Mountain Conference on Iterative Methods, Copper Mountain, Colorado, April 9–14 1992.

[6] ——, *Solution of large, sparse systems of linear equations in massively parallel applications*, Preprint MCS-P313-0692, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.

[7] ——, *A parallel graph coloring heuristic*, SIAM Journal on Scientific and Statistical Computing, 14 (1993).

[8] V. S. Sunderam, *PVM: A framework for parallel distributed computing*, Concurrency: Practice and Experience, 2 (1990), pp. 315–339.