RC 21888 (98472) November 20, 2000
Computer Science/Mathematics

IBM Research Report

WSMP: Watson Sparse Matrix Package
Part II – direct solution of general sparse systems
Version 1.0.0β

Anshul Gupta

IBM Research Division
T.J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

anshul@watson.ibm.com

WSMP: Watson Sparse Matrix Package
Part II – direct solution of general sparse systems

Version 1.0.0β*

Anshul Gupta

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

anshul@watson.ibm.com

IBM Research Report RC 21888 (98472)

November 20, 2000

*Since this document describes software that might be upgraded from time to time, *WSMP* users and the readers of this report are encouraged to request the latest versions by sending e-mail to *anshul@watson.ibm.com*. The software was last updated on November 21, 2000. The documentation was last updated on November 21, 2000

Contents

1 Introduction to Part II

The Watson Sparse Matrix Package, *WSMP*, is a high-performance, robust, and easy to use software package for solving large sparse systems of linear equations on the IBM RS6000 workstations. *WSMP* is comprised of two parts, both of which are bundled in the same library. This document describes Part II of the package, which pertains to the solution of *general* sparse systems of linear equations. Part I of the *WSMP* library [3], which deals with the solution of *symmetric* sparse systems of linear equations, is described in another document that can be obtained from *ftp://ftp.cs.umn.edu/users/kumar/anshul/wsmp1.ps* via anonymous FTP or through your web browser. Technical papers related to the software and the example programs in Appendices A and B can be obtained from the site *ftp://ftp.cs.umn.edu/users/kumar/anshul*.

For solving general sparse systems, *WSMP* uses a modified version of the multifrontal algorithm [1] for matrices with an unsymmetric pattern of nonzeros. *WSMP* supports threshold partial pivoting for general matrices with a user defined threshold. *WSMP* automatically exploits SMP parallelism on an RS6000 workstation or SP node with multiple CPUs and this parallelism is transparent to the user. On an RS6000 with four 375 MHz Power3 CPUs, *WSMP* has been clocked at up to 2.4 GFLOPS for factoring general sparse matrices with partial pivoting. A detailed comparison of various general sparse solver packages, including the serial version of *WSMP*, can be found in [6].

Currently, *WSMP* does not support the solution of general/unsymmetric sparse systems in a message-passing parallel environment. *WSMP* does not have out-of-core capabilities and the problems must fit in the main memory for reasonable performance.

The organization of this document is as follows. Section 2 lists the various libraries that are available and describes how to obtain and link the libraries. Section 3 describes the changes in the latest version that may affect the users of earlier versions. Section 4 gives an overview of the functionality of *WSMP* for solving general sparse systems. Section 5 gives a detailed description of the main serial/multithreaded routine that provides an advanced single routine interface to the entire software. This section also describes the input data structures for the serial and multithreaded cases. Section 6 describes user callable routines that provide a simpler interface to the serial and multithreaded solver, but omit some of the advanced features. Section 7 describes a few utility routines available to the users to provide some extra flexibility in memory management and data distribution. Section 8 contains the terms and conditions that all users of the package must adhere to.

2 Obtaining, Linking, and Running with *WSMP*

Both serial and multithreaded routines are a part of the *WSMP* library, which can be used on single-CPU or SMP workstations. In addition to Sections 2.3 and 2.5, please refer to Appendices A and B for examples on using the *WSMP* library.

2.1 Compiler and OS requirements

The current version of *WSMP* requires that the following be installed on your system:

1. XLF Fortran compiler version 6.1.0.3 or higher.

2. XLC C language compiler version 4.4.0.0 or higher.

3. AIX version 4.3.2 or higher.

Note 2.1 *All libraries are compiled in 32-bit mode. Therefore, any code using the libraries must be compiled in 32-bit mode, even on the 64-bit architectures, such as Power3.*

2.2 Obtaining the software

The serial/multithreaded *WSMP* library has two versions: *libwsmp.a* for all RS6000 workstation and *libwsmpp3.a* compiled and optimized specifically for the Power3 architecture.

The libraries can be obtained by sending e-mail to *anshul@watson.ibm.com* with the subject line containing the library name followed by one or more blanks followed by the e-mail address at which the appropriate library and the accompanying documentation will be e-mailed. The body of the message must contain the requester's name, affiliation, and the intended use of the software. The users must read Section 8 of this document before using the software. The documentation is sent in the postscript format and the libraries are e-mailed as uuencoded files, which should yield binary files with *.a* extensions upon successful decoding. You may need to run *ranlib* on the decoded files. If you are using older versions of XLF or AIX than the ones for which the libraries are available, please send e-mail to *anshul@watson.ibm.com* with your AIX and XLF versions and we will attempt to make a library suitable for your system.

2.3 Linking the libraries

The *WSMP* library contains code written in both C and FORTRAN. The libraries can be linked with either C or FORTRAN programs using either *xlf* or *xlc* commands (or their variations).

Here are some examples of linking the user source or object files with the *WSMP* libraries to create an executable file. The user may need to use other flags and options as required by the user's code.

xlf_r <source/object files> -o <executable target> -lpthreads -lm_r -lwsmpp3 -L<path to parent directory of *libwsmpp3.a*>

xlc_r <source/object files> -o <executable target> -lpthreads -lm_r -lxlf90_r -lwsmpp3 -L<path to parent directory of *libwsmpp3.a*>

Note 2.2 *On most machines, you will need to use the* -bmaxdata:<*membytes*> *option while linking so that the data segment of the program can use all the available memory. Here* membytes *is the integer number of bytes of memory that you want your program to be able to allocate. If you get an error code which, modulo 1000, is -102 (see description of* IPARM(64) *in Section 5.3.9 for more details) when you are not expecting the program to run out of memory, then using this option or increasing* membytes *may solve the problem.*

Note 2.3 *Note that we used the reentrant versions (with _r suffix) of the various libraries. This is because most RS6000 systems are shared-memory multiprocessors and reentrant routines must be used in the multithreaded environment.*

Note 2.4 *Although* WSMP *library contains multithreaded code, the library itself is* **not** *thread-safe. Therefore, the calling program cannot invoke multiple instances of the routines contained in* WSMP/PWSMP *from different threads at the same time.*

Note 2.5 *Note that the XL FORTRAN library* xlf90_r *is required if* xlc_r *command is used for creating the executable. The math library should also be linked in by adding* -lm_r *to the linking command as shown above.*

2.4 Using WSMP in conjunction with other IBM software

If you are linking your program with other IBM libraries, such as, *ESSL*, *PESSL*, or *OSL*, etc. along with *WSMP*, then you may encounter occasional problems due to interference between these libraries. Such problems can be avoided by obeying *both* of the following simple rules:

1. At the time of linking, always link the *WSMP* and *PWSMP* libraries before any other libraries. For example "xlf_r foo.f -lwsmp -lessl" is the recommended way of using both *ESSL* and *WSMP* together.

2. *WSMP* performs self initialization when the first call to any user-callable *WSMP* routine is made. However, due to some common non-user-callable routines between *WSMP* and other IBM libraries, you may get a run time error if such a routine is called before *WSMP* has been initialized. A call to *WSMP_INITIALIZE()* (Section 7.1) at the beginning of your program (before any library call) eliminates the possibility such interference.

2.5 Setting environment variables and running

In a shared memory environment, the user may want to experiment with setting different values for YIELDLOOPTIME and SPINLOOPTIME to see which values yield the best performance on their system. Generally recommended values are YIELDLOOPTIME=500 and SPINLOOPTIME=500, but optimal values may vary depending on the type of machine and the number of CPUs.

2.6 Linking and execution-time problems

Please refer to Section 2.3 for instructions on linking the *WSMP* library. The most likely cause of any problems encountered during linking or loading would be an incompatible version of the operating system or the compiler libraries. Also, please read Section 2.4 carefully to avoid problems when using *WSMP* with other IBM libraries.

WSMP routines perform minimal input argument error-checking and it is the user's responsibility to call *WSMP* subroutines with correct arguments, especially, valid options and matrices. In case of an invalid input, it is not uncommon for a routine to hang or to crash with segmentation fault. In the parallel version, on rare occasions, insufficient memory can also cause a routine to hang or crash before all the processes have a chance to return safely with an error report; however, such occurrences are rare.

Unlike the input argument and memory related errors, the numerical error checking capabilities of the computational routines are quite robust.

Version 1.0 of *WSMP* is a beta version. Any bugs or other problems with the software or this manual may be reported via e-mail to *anshul@watson.ibm.com*.

3 Changes from Earlier Versions

Version 1.0 of *WSMP* is identical in functionality to version 4.6 of *WSSMP* [5] for solving symmetric sparse systems. Version 1.0 of *PWSMP* does not include routines for solving general sparse systems; so currently unsymmetric systems can be solved only in serial or shared-memory modes.

4 Overview of Functionality

WGSMP is the primary routine in *WSMP* for solving general sparse systems of linear equations and is described in detail in Section 5. Additionally, *WSMP* contains some routines that provide a simpler interface to the solver (see Section 6 for more details). Both interfaces allow the users to perform any appropriate subset of the following tasks: (1) Analysis and reordering, (2) LU factorization, (3) Back substitution, and (4) Iterative refinement. When using the *WGSMP* routine, *IPARM(2)* and *IPARM(3)* control the subset of the tasks to be performed. When using the simple interfaces, the tasks or the subsets of tasks to be performed are determined by the name of the routine.

In the following subsections, we describe the key functions and the interdependencies of the four tasks mentioned above.

4.1 Analysis and reordering

The analysis phase generates permutations for the rows and columns of the input matrix. These permutations are designed to minimize fill during factorization and to provide ample parallelism and load-balance during message-passing or multithreaded parallel factorization. Additionally, this phase takes the numerical values in the matrix into account too and uses certain heuristics to generate permutations that would minimize partial pivoting during numerical factorization. Therefore, it is necessary to pass the entire matrix, along with the numerical values, to the analysis phase. The original matrix is not altered at this stage; the permutations are stored and used internally.

This phase also performs symbolic factorization based on the row and column permutations it generates and estimates the computational and memory requirements of the numerical phases to follow. Of course, these are only estimates because the actual computational and memory requirement of LU factorization depends on the sequence of pivots chosen during factorization to ensure numerical stability.

If an application involves solving several systems with coefficient matrices of identical nonzero structure but different numerical values, then the analysis and reordering step needs to performed only for the first matrix in the sequence. For the subsequent systems, only factorization and triangular solution (and iterative refinement, if required) need to be performed. Although the analysis phase takes numerical values into account, the software adapts to the changing numerical values in the matrix (as long as the structure is identical to the one used in analysis), and therefore, the analysis phase needs to be performed only once for matrices with the same structure but different numerical values. Please refer to the description of *IPARM(27)* for more details.

4.2 LU factorization

Once the analysis step has been performed, numerical factorization can be called any number of times for matrices with identical nonzero pattern (determined by *IA* and *JA*) but possibly different numerical values in *AVALS*. The matrices L and U that are produced as a result of LU factorization are stored internally and are not directly available to the user. *WSMP* uses these matrices for triangular solve(s) that follow factorization.

LU factorization in *WSMP* uses threshold pivoting and can use either a user provided threshold or a threshold that it generates internally depending on the degree of diagonal dominance of the input matrix. This threshold α is a double precision value between 0.0 and 1.0. At the beginning of the i-th pivoting step, let d be the absolute value of the diagonal entry (i.e., $d = |a_{i,i}|$ and r be the maximum absolute value of any entry in the i-th column below the diagonal. Let this entry belong to row j ($j \geq i$, $r = |a_{j,i}|$). Now if $d \geq \alpha r$, then no row exchange is performed and the i-th row is used as the pivot row. However, if $d < \alpha r$, then row i can be exchanged with any row k ($k > i$), such that the absolute value s of the k-th entry in column i is greater than or equal to αr. Note that this is somewhat different from traditional partial pivoting, according to which, rows i and j would have been exchanged if $d \geq \alpha r$. In *WSMP*, we chose the pivot row that satisfies the threshold criterion and is likely to cause the least fill-in.

4.3 Back substitution

The back substitution or the triangular solve phase generates the actual solution to the system of linear equations. This phase uses the internally stored factors generated by a previous call to numerical factorization. The user can solve multiple systems together by providing multiple right-hand sides, or can solve for multiple instances of single or multiple right-hand sides one after the other. If systems with multiple right-hand sides need to be solved and all right-hand sides are available together, then solving them all together is significantly more efficient than solving them one at a time.

WSMP keeps track of all permutations affected by the fill-reducing ordering and due to partial pivoting internally. The user presents the RHS vector in the same order as the original row ordering of the input coefficient matrix and obtains the solution in the same order too.

4.4 Iterative refinement

Iterative refinement can be used to improve the solution produced by the back-substitution phase. Often, it is cheaper to specify a low pivoting threshold, which may result in a faster (but less accurate) factorization due to fewer row-exchanges, and to recover the accuracy via iterative refinement. As a part of iterative refinement, the backward error is also computed, which is available to the user as an output. The option of using extended precision arithmetic for iterative refinement is available.

5 The Primary Serial Subroutine: *WGSMP*

This section describes the use of the *WGSMP* subroutine and its calling sequences in detail. There are four basic tasks that *WGSMP* is capable of performing, namely, *analysis and reordering*, *LU factorization*, *forward and backward solve*, and *iterative refinement* (see Note 5.4). The same routine can perform all or any number of these functions in sequence depending on the options given by the user via parameter *IPARM* (see Section 5.3). In addition, a call to *WGSMP* can be used to get the default values of the options without any of the five basic tasks being performed. See the description of *IPARM(1)*, *IPARM(2)*, and *IPARM(3)* in Section 5.3.9 for more details.

In addition to the advanced interface that the *WSMP* library provides via the single subroutine *WGSMP*, there are a number of other subroutines that provide a simpler interface. These subroutines are described in detail in Section 6.

5.1 Types of matrices accepted and their input format

The *WGSMP* routine works for any non-singular square sparse matrix. Even if the original matrix is symmetric, *WGSMP* expects the entire sparse matrix as input. All floating point values must be 8-byte real numbers and all integers must be 4 bytes long. Currently, two input formats are supported, namely, compressed sparse rows (CSR) and compressed sparse columns (CSC). Figure 1 illustrates both input formats; they are also explained briefly in Sections 5.3.2, 5.3.3, and 5.3.4.

WGSMP supports both C-style indexing starting from 0 and FORTRAN-style indexing starting from 1. Once a numbering style is chosen, all data structures must follow the same numbering convention which must stay consistent through all the calls referring to a given system of equations. Please refer to the description of *IPARM(5)* in Section 5.3.9 for more details.

5.2 Pivoting options

By means of *IPARM(8..12)*, *DPARM(11..12)*, and *DPARM(22)*, a user can customize the way *WGSMP* selects pivots for elimination in LU factorization. Please refer to the detailed description of these parameters in Sections 5.3.9 and 5.3.10. Some of the commonly used scenarios are presented here:

- **No pivoting:** Certain systems of linear equations do not require partial pivoting and the factorization is stable with the input sequence of rows. For such systems, *IPARM(8..12)* must be set to (0,0,0,0,0) to avoid the overhead of unnecessary pivoting. If the matrix is poorly scaled, then setting *IPARM(8..12)* to (0,1,0,0,0) will perform equilibration without pivoting during factorization.

- **Threshold pivoting with user-supplied threshold:** For systems requiring partial pivoting, it is recommended that *IPARM(8..10)* be set to (1,0,1). Although these options do not directly affect pivoting, together they transform the matrix such that the magnitude of each diagonal entry is 1.0 and that of any nondiagonal entry is less than or equal to 1.0. Such transformation has shown to significantly reduce the cost of pivoting during factorization. The recommended values of *IPARM(11..12)* are (1,0). The pivoting threshold, which is a double precision value greater than 0.0 and less than or equal to 1.0, must be placed in *DPARM(11)*. The default values for *IPARM(8..12)* in *WGSMP* are (1,0,1,1,0); i.e., by default, *WGSMP* is set to run in this mode. The default value of pivoting threshold *DPARM(11)* is 0.05.

  Note 5.1 WGSMP *uses several mechanisms, other than partial pivoting, to enhance the accuracy of the final solution. These include a static permutation of rows maximize the diagonal product [7, 9, 2, 8], scaling, and iterative refinement in double and quadruple precision. Therefore, it is recommended that the smallest pivoting threshold that yields a solution with acceptable accuracy should be used. Minimizing row-interchanges associated with partial pivoting saves time and memory.*

- **Threshold pivoting with WGSMP-supplied threshold:** If you are not sure what pivoting threshold to use, then you can place 0.0 in *DPARM(11)* and *WGSMP* will figure out an appropriate threshold to used

	1	2	3	4	5	6	7	8	9
1	14.		-5.				-1.	-6.	
2		14.	-1.		-3.				-1.
3	-1.		16.				-2.		
4				14.				-3.	
5					14.	-1.			-1.
6		-2.		-1.		16.	-2.	-4.	
7				-1.			16.		
8	-3.		-4.		-3.		-4.	71.	
9		-1.	-2.					-4.	16.

A 9 X 9 general sparse matrix.

The storage of this matrix in the input formats accepted by WGSMP is shown in the table below:

	CSC Format			CSR Format		
K	IA(K)	JA(K)	AVALS(K)	IA(K)	JA(K)	AVALS(K)
1	1	1	14.0	1	1	14.0
2	4	3	-1.0	5	3	-5.0
3	7	8	-3.0	9	7	-1.0
4	12	2	14.0	12	8	-6.0
5	15	6	-2.0	14	2	14.0
6	17	9	-1.0	17	3	-1.0
7	19	1	-5.0	22	5	-3.0
8	24	2	-1.0	24	9	-1.0
9	30	3	16.0	29	1	-1.0
10	33	8	-4.0	33	3	16.0
11		9	-2.0		7	-2.0
12		4	14.0		4	14.0
13		6	-1.0		8	-3.0
14		7	-1.0		5	14.0
15		2	-3.0		6	-1.0
16		5	14.0		9	-1.0
17		8	-3.0		2	-2.0
18		5	-1.0		4	-1.0
19		6	16.0		6	16.0
20		1	-1.0		7	-2.0
21		3	-2.0		8	-4.0
22		6	-2.0		4	-1.0
23		7	16.0		7	16.0
24		8	-4.0		1	-3.0
25		1	-6.0		3	-4.0
26		4	-3.0		5	-3.0
27		6	-4.0		7	-4.0
28		8	71.0		8	71.0
29		9	-4.0		2	-1.0
30		2	-1.0		3	-2.0
31		5	-1.0		8	-4.0
32		9	16.0		9	16.0

Figure 1: Illustration of the two input formats for the serial and multithreaded *WGSMP* routines.

based on the input matrix. *IPARM(8..12)* should be used just as above; i.e, it is recommended that they are set to (1,1,1,1,0).

- **Static pivoting:** Dynamic row exchanges that accompany threshold pivoting can have an adverse impact on the performance of LU factorization, significantly so in parallel. Therefore, *WGSMP* provides users the option to use only static pivoting (i.e., a static row ordering) that maximizes the diagonal entries in the matrix relative to the non-diagonal entries before starting the factorization. This minimizes the occurrence of small pivots during factorization, which may compromise the stability of the solution. To use static pivoting, *IPARM(8..12)* should be set to (1,1,1,0,1). On the rare occasions when a pivot is indeed too tiny (as defined by the user by the input threshold in *DPARM(12)*), then *WGSMP* will solve a slightly perturbed system such that the pivot growth for that step does not exceed *DPARM(22)*. Please refer to the descriptions of *IPARM(12), DPARM(12)*, and *DPARM(22)* for more details. It is recommended that *DPARM(22)* be set to the square root of the machine precision (approx 2×10^{-8}). In most cases, the loss of accuracy due to the perturbation can be recovered by iterative refinement [8].

- **Row and Column pivoting:** The default method of pivoting in *WGSMP* (activated by *IPARM(8..12)* = (1,0,1,1,0)) chooses a pivot row such that the diagonal element is larger in magnitude than the product of the largest element in the pivot column multiplied by the pivoting threshold. The magnitude of the diagonal element is not checked with respect to other elements in the pivot row. This method works for most sparse systems. However, in some cases, the resulting growth along the rows may yield unacceptable accuracy. If increasing the pivoting threshold does not bring the accuracy into an acceptable range, then the user may need to turn both row and column pivoting on by setting *IPARM(28)* to 1. The default value of *IPARM(28)* is 0. Usually, in order to make row and column pivoting effective, block-triangulation needs to be suppressed by setting *IPARM(21)* to 0, so that all elements of the matrix can participate in pivot selections. To summarize, some sparse systems are really tough to solve, and may require one or more of the following actions on part of the user in addition to using the default pivoting options: (1) increasing the pivoting threshold *DPARM(11)*, (2) suppressing block-triangulation by setting *IPARM(21)* to 0, (3) switching to row and column pivoting from a simple row pivoting by setting *IPARM(28)* to 1. All these actions have the potential of slowing down LU factorization considerably and must be used judiciously.

The modes described above are some of the common ones that a user might use, but these are not the only possible scenarios. For example, a user may chose to use partial pivoting, but switch off the prepermutation to a heavy-diagonal form by setting *IPARM(8..12)* to (0,0,0,1,0). Similarly, one can use a combination of threshold pivoting and perturbation by setting both *IPARM(11)* and *IPARM(12)* to 1. If a pivot is too tiny (as determined by *DPARM(11)*), then a row-interchange is performed. Otherwise, for the pivots that are not too close to zero, but are still small enough (as determined by *DPARM(12)*), the pivot magnitude is artificially increased (perturbed) and computation proceeds without a row-interchange. Please refer to the description of *IPARM(12)* in Section 5.3.10 for more details.

5.3 Calling sequence of the WGSMP subroutine

There are four types of arguments, namely input (type $\mathbf{I}$), output (type $\mathbf{O}$), modifiable (type $\mathbf{M}$), and reserved (type $\mathbf{R}$). The input arguments are read by *WGSMP* and remain unchanged upon execution, the output arguments are not read but some useful information is returned via them, the modifiable arguments are read by *WGSMP* and modified to return some information, the reserved arguments are not read but their contents may be overwritten by unpredictable values during execution. The reserve arguments may change to one of the other types of arguments in the future serial and parallel releases of this software.

In the remainder of this document, the "system" refers to the sparse linear system of N equations of the form $Ax = b$, where A is a general sparse coefficient matrix of dimension N, b is the right-hand-side vector/matrix and x is the solution vector/matrix, whose approximation $\overline{x}$ computed by *WGSMP* overwrites b when *WGSMP* is called to compute the solution of the system. The example program in Appendix A illustrates the use of the *WGSMP* subroutine for the matrix shown in Figure 1.

Note 5.2 *Recall that* WGSMP *supports both C-style (starting from 0) and FORTRAN-style (starting from 1) numbering. The description in this section assumes FORTRAN-style numbering and C users must interpreted it accordingly. For example,* IPARM(11) *will actually be* IPARM[10] *in a C program calling* WGSMP.

Note 5.3 *The original code for* WGSMP *is in FORTRAN and expects the parameters to be passed by reference. Therefore, when calling* WGSMP *from a C program, the addresses of the parameters described in Section 5.3 must be passed.*

The calling sequence and description of the parameters of *WGSMP* is as follows. When an input data structure is not accessed in a particular call, a NULL pointer or any scalar can be passed as a place holder for that argument. The analysis (ordering and symbolic factorization) phase of *WGSMP* accesses and uses *AVALS*—something that most conventional sparse solvers don't do.

WGSMP (N, IA, JA, AVALS, B, LDB, NRHS, DARRAY, IPARM, DPARM)

5.3.1 N (type I): matrix dimension

```
INTEGER N
```

This is the number of rows and columns in the sparse matrix A or the number of equations in the sparse linear system $Ax = b$.

5.3.2 IA, (type I): row (column) pointers

```
INTEGER IA (N + 1)
```

IA is an integer array of size one greater than N. *IA(I)* points to the first column (row) index of row (column) I in the array *JA* in CSR (CSC) format. Refer to Figure 1 and description of *IPARM(4)* in Section 5.3.9 for more details.

5.3.3 JA, (type I): column indices

```
INTEGER JA ( * )
```

The integer array *JA* contains the column (row) indices of the sparse matrix A stored in CSR (CSC) format. The column (row) indices of each row (column) must follow the indices of the previous column (row). Moreover, the column (row) indices should sorted in increasing order. *WSMP* provides two utility routines to sort the indices (see Section 7 for details).

5.3.4 AVALS, (type I or M): nonzero values of the coefficient matrix

```
DOUBLE PRECISION AVALS ( * )
```

The array *AVALS* contains the actual double precision values corresponding to the indices in *JA*. The size of *AVALS* is the same as that of *JA*. See Figure 1 for more details.

5.3.5 B, (type M): right-hand side vector/matrix

```
DOUBLE PRECISION B ( LDB, NRHS )
```

The $N\times$ *NRHS* dense matrix B (stored in an *LDB* $\times$ *NRHS* array) contains the right-hand side of the system of equations $Ax = b$ to be solved. If the number of right-hand side vectors, *NRHS*, is one, then B can simply be a vector of length N. B is accessed only in the triangular solution and iterative refinement phases.

5.3.6 LDB, (type I): leading dimension of B

```
INTEGER LDB
```

LDB is the leading dimension of the right-hand side matrix if *NRHS* > 1. *LDB* must be greater than or equal to N. Even if *NRHS* $= 1$, *LDB* must be greater than 0. *LDB* is accessed only when B is accessed.

5.3.7 NRHS, (type I): number of right-hand sides

```
INTEGER NRHS
```

NRHS is the second dimension of B; it is the number of right-hand sides that need to be solved for.

5.3.8 DARRAY, (type O): double precision output info

```
DOUBLE PRECISION DARRAY ( N, NRHS )
```

If *IPARM(25)* is 0, then *DARRAY* is not accessed. If *IPARM(25)* is 1 on input, then on return from iterative refinement, *DARRAY(I,J)* is set to to the I-th component of the backward error while solving for the J-th RHS.

Note that the user needs to provide a valid double precision array of size $N \times NRHS$ only if *IPARM(25)* is set to 1 on input; otherwise, *DARRAY* can just be a placeholder double precision pointer. *DARRAY* is accessed only in the triangular solution and iterative refinement phases.

5.3.9 IPARM, (type I, O, M, and R): integer array of parameters

```
INTEGER IPARM ( 64 )
```

IPARM is an integer array of size 64 that is used to pass various optional parameters to *WGSMP* and to return some useful information about the execution of a call to *WGSMP*. If *IPARM(1)* is 0, then *WGSMP* fills *IPARM(4)* through *IPARM(64)* and *DPARM* with default values and uses them. The default initial values of *IPARM* and *DPARM* are shown in Table 1. *IPARM(1)* through *IPARM(3)* are mandatory inputs, which must always be supplied by the user. If *IPARM(1)* is 1, then *WGSMP* uses the user supplied entries in the arrays *IPARM* and *DPARM*. Note that some of the entries in *IPARM* and *DPARM* are of type M or O. It is possible for a user to call *WGSMP* only to fill *IPARM* and *DPARM* with the default initial values. This is useful if the user needs to change only a few parameters in *IPARM* and *DPARM* and needs to use most of the default values. Please refer to the description of *IPARM(2)* and *IPARM(3)* for more details. Note that there are no default values for *IPARM(2)* and *IPARM(3)* and these must always be supplied by the user, whether *IPARM(1)* is 0 or 1.

- **IPARM(1), type I or M:**

  If *IPARM(1)* is 0, then the remainder of the *IPARM* array and the *DPARM* array are filled with default values by *WGSMP* before further computation and *IPARM(1)* itself is set to 1. If *IPARM(1)* is 1 on input, then *WGSMP* uses the user supplied values in *IPARM* and *DPARM*.

- **IPARM(2), type M:**

  On input, *IPARM(2)* must contain the number of the starting task. On output, *IPARM(2)* contains 1 + number of the last task performed by *WGSMP*, if any. This is to facilitate users to restart processing on a problem from where the last call to *WGSMP* left it. If *WGSMP* performs no task, then, on output, *IPARM(2)* is set to *max(IPARM(2),IPARM(3)+ 1)*. *WGSMP* can perform any set of consecutive tasks from the following list:

Task # 1:	Analysis and Reordering
Task # 2:	LU Factorization
Task # 3:	Forward and Backward Elimination
Task # 4:	Iterative Refinement

	IPARM			DPARM		
Index	Default	Description	Type	Default	Description	Type
1	mandatory I/P	default/user defined	M	-	unused	-
2	mandatory I/P	starting task	M	-	unused	-
3	mandatory I/P	last task	I	-	unused	-
4	0	I/P format	I	-	largest pivot	O
5	1	numbering style	I	-	smallest pivot	O
6	5	max. # iter. refs.	M	5.e-16	back err. lim.	I
7	3	residual norm type	I	-	backward error	O
8	1	max. matching use	I	-	unused	-
9	0	scaling w/o matching	I	-	unused	-
10	1	scaling w/ matching	I	-	unused	-
11	1	thresh. pivoting opt.	I	0.05	tiny piv. thresh.	I
12	0	pivot perturb. opt.	I	2.e-8	small piv. thresh.	I
13	-	# row/col ex.	O	-	unused	-
14	-	# perturbations	O	-	unused	-
15	0	(# factor)/(# anal.)	I	-	unused	-
16	1	ordering option 1	I	-	unused	-
17	0	ordering option 2	I	-	unused	-
18	0	ordering option 3	I	-	unused	-
19	0	ordering option 4	I	-	unused	-
20	0	ordering option 5	I	-	unused	-
21	1	block triangular form	I	-	unused	-
22	-	# blocks in B.T.F.	O	2.e-8	small piv. repl.	I
23	-	actual $NNZ_L + NNZ_U$	O	-	actual fact. ops.	O
24	-	symbolic $NNZ_L + NNZ_U$	O	-	symbolic fact. ops.	O
25	0	DARRAY use	I	-	unused	-
26	-	# iter. ref. steps	O	-	unused	-
27	0	# fact. before re-analyze	I	1.0	re-analyze condition	-
28	0	row and col. pivoting	I	-	unused	-
29	0	garbage collection	I	-	unused	-
30-63	-	reserved	R	-	reserved	R
64	-	info O/P	O	-	unused	-

Table 1: The default initial values of the various entries in *IPARM* and *DPARM* arrays. A '-' indicates that the value is not read by *WGSMP*. Please refer to the text for details on ordering options *IPARM(16:20)*. (# $\equiv$ "number of").

Note 5.4 WGSMP *can process only one matrix at a time. A user cannot factor one matrix, then factor a second matrix, and then solve a system using the first factor. In other words,* WGSMP *can work on a system in increasing order of task numbers. If a call to* WGSMP *is made with a starting task number in* IPARM(2) *that is less than or equal to the number of the last task performed by* WGSMP *in a previous call, then the results of the previous call are lost.*

- **IPARM(3), type I:**

  IPARM(3) must contain the number of the last task to be performed by *WGSMP*. In a call to *WGSMP*, all tasks from *IPARM(2)* to *IPARM(3)* are performed (both inclusive). If *IPARM(2)* $>$ *IPARM(3)* or both *IPARM(2)* and *IPARM(3)* is out of the range 1–4, then no task is performed. This can be used to fill *IPARM* and *DPARM* with default values; e.g., by calling *WGSMP* with *IPARM(1)* $= 0$, *IPARM(2)* $= 0$, and *IPARM(3)* $= 0$.

- **IPARM(4), type I:**

  IPARM(4) denotes the format in which the coefficient matrix A is stored. *IPARM(4)* $= 0$ denotes CSR format and *IPARM(4)* $= 1$ denotes CSC format. The default is CSR. Both formats are illustrated in Figure 1.

- **IPARM(5), type I:**

  If *IPARM(5)* $= 0$, then C-style numbering (starting from 0) is used; If *IPARM(5)* $= 1$, then FORTRAN-style numbering (starting from 1) is used. In C-style numbering, the matrix rows and columns are numbered from 0 to $N-1$ and the indices in *IA* should point to entries in *JA* starting from 0. *IPARM(5)* $= 1$ is the default.

- **IPARM(6), type M:**

  On input to the iterative refinement step, *IPARM(6)* should be set to the maximum number of steps of iterative refinement to be performed. Also refer to the description of *IPARM(7)* and *DPARM(6)* for more details. *DPARM(6)* provides a means of performing none or fewer than *IPARM(6)* steps of iterative refinement if a satisfactory level of accuracy of the solution (in terms of backward error) has been achieved. Upon returning from iterative refinement, *IPARM(26)* contains the actual number of refinement steps performed.

- **IPARM(7), type I:**

  If *IPARM(7)* $= 0$, 1, 2, or 3, then the residual in iterative refinement is computed in double precision (the same as the remainder of the computation). If *IPARM(7)* $= 4$, 5, 6, or 7, then the residual in iterative refinement is computed in quadruple precision (which is twice the precision of the remainder of the computation). If *IPARM(7)* $= 0$ or 4, then exactly *IPARM(6)* number of iterative refinement steps are performed without checking for the backward error. If *IPARM(7)* $= 1$, 2, 3, 5, 6, or 7, then iterative refinement is performed until the number of iterative refinement steps is equal to *IPARM(6)* or until the the backward error given by $\frac{\|b-A\overline{x}\|}{\|b\|}$ falls below the input value in *DPARM(6)*. Here A is the coefficient matrix, $\overline{x}$ is the computed solution, and b is the right-hand side vector. If *IPARM(7)* $= 1$ or 5, then 1-norms are used in computing the backward error, if *IPARM(7)* $= 2$ or 6, then 2-norms are used, and if *IPARM(7)* $= 3$ or 7, then infinity-norms are used. Moreover, if *IPARM(7)* $= 1$, 2, 3, 5, 6, or 7, then the actual backward error at the end of the last iterative refinement step is placed in *DPARM(7)*.

  If *NRHS* > 1, then the maximum of the backward errors amongst the *NRHS* solution vectors is considered. Also note that, if scaling is performed (i.e., if *IPARM(10)* $= 0$), then the backward errors are computed with respect to the scaled system and not the original system.

- **IPARM(8), type I:**

  WGSMP uses a maximum weight matching on the bipartite graph induced by the sparse coefficient matrix to permute its row such that the product of the absolute values of the diagonal is maximized [9, 2, 8, 7]. By default, indicated by *IPARM(8)* $= 1$, this option is turned on. *IPARM(8)* should be set to 0 in order to turn this option off. Turning it off may be useful if LU factorization of the original matrix is known to be stable without partial pivoting. If the original matrix is symmetric and has a stable factorization without pivoting, then it is important to turn off the maximum-matching based prepermutation because it may destroy the symmetry and make factorization unnecessarily costlier.

- **IPARM(9), type I:**

  If *IPARM(8)* is 1, then during the analysis and reordering phase, *WGSMP* uses a maximum bipartite matching algorithm to permute the rows such that the product of the absolute values of the diagonal entries is maximized. In addition to a row permutation, the maximum matching algorithm also produces vectors for scaling the rows and the columns of the sparse matrix such that the magnitude of each diagonal entry of the scaled matrix is 1.0. If a maximum matching is not performed (for instance, to preserve structural sparsity), then by setting *IPARM(9)* to 1, a simple equilibration can still be performed on poorly scaled matrices to make the magnitude of each diagonal entry equal to 1.0. The default value of *IPARM(9)* is 0. Note that the scaling using the vectors produced by the maximum matching algorithm (which is switched on by setting *IPARM(10)* to 1) is usually more effective, and *IPARM(9)* should be set to 1 only when maximum matching is not used.

- **IPARM(10), type I:**

  An input of *IPARM(10)* $= 0$ during numerical factorization implies that *WGSMP* will not perform a scaling of the input matrix using the vectors generated by applying the maximum bipartite matching algorithm to the input matrix. *IPARM(10)* $= 1$, which is the default, implies that such scaling is performed in an attempt to improve the numerical stability of factorization. Please refer to the description of *IPARM(8)* for more details.

- **IPARM(11), type I:**

  IPARM(11) and *IPARM(12)* instruct *WGSMP* how to handle small or zero pivots. If *IPARM(11)* is 0, then no row exchanges are performed during factorization. The computation will proceed unless a zero diagonal entry is encountered, in which case, either an artificial nonzero value is placed at the diagonal depending on *IPARM(12)* and *DPARM(12)*, or the corresponding row/column number is reported in *IPARM(64)* and factorization stops. Please refer to the description of *IPARM(12)* for more details on the actions that *WGSMP* might take if *IPARM(11)* is 0.

  Let α be the pivoting threshold ($0.0 < \alpha \leq 1.0$). Then setting *IPARM(11)* $= 1$ before a *WGSMP* call for factorization ensures that the pivot growth does not exceed $1/\alpha$ at any elimination step. Let d be the absolute value of the diagonal entry just before the i-th elimination step. Let r be the maximum absolute value among all entries in the i-th column. If *IPARM(11)* is 0, then no row exchanges are performed and LU factorization proceeds normally if $d \geq \alpha r$ (see description of *IPARM(12)*, though). However, if $d < \alpha r$, then the i-th row can be exchanged by any row such that the absolute value of the entry in the i-th column of that row is greater than or equal to αr. If all entries in column i are zero (i.e., the matrix is singular), then the factorization is terminated and i is returned in *IPARM(64)*. A numbering from 1 to N is used to indicate this kind of failure, even if the input uses C-style numbering.

  The pivoting threshold α is equal to *DPARM(11)* if *DPARM(11)* > 0.0 on input (i.e., the user supplies the threshold). If *IPARM(11)* $= 1$ and *DPARM(11)* $= 0.0$, then *WGSMP* choses an appropriate threshold. The input in *DPARM(11)* must always be non-negative.

  The default value of *IPARM(11)* is 1.

  Note 5.5 *The input in* IPARM(11) *must be set before the analyze phase so that it knows that partial pivoting is intended during numerical factorization. If the value in* IPARM(11) *is different during the analyze and factor phases, then the program may crash or generate incorrect results.*

- **IPARM(12), type I:**

  IPARM(12) $= 0$, which is the default, has no effect. If *IPARM(12)* $= 1$, then α and β are chosen and used as follows:

If *IPARM(11)* $= 1$, then α is the the user-defined or *WGSMP*-defined pivoting threshold determined by the input *DPARM(11)*. If *IPARM(11)* $= 0$, then $\alpha = 0.0$. β is the input value in *DPARM(12)*. Let d be the absolute value of the diagonal entry $a_{i,i}$ just before the i-th elimination step. Let r be the maximum absolute value among all entries in i-th column. If $\alpha r \leq d < \beta r$, then $a_{i,i}$ is replaced by *sign(*$a_{i,i}$*)* $\times r \times$ *DPARM(22)* and factorization proceeds with the new value of $a_{i,i}$.

- **IPARM(13), type O:**

  After factorization, *IPARM(13)* contains the total number of row and column interchanges performed as a result of partial pivoting. The maximum possible value of *IPARM(13)* on output can be $2N - 2$, because each pivot selection, except the last one, can entail both a row and column interchange.

- **IPARM(14), type O:**

  After factorization, *IPARM(14)* contains the number of diagonal entries that were perturbed in order to contain pivot growth. The perturbation, if any, is controlled by the user inputs in *IPARM(12)* and *DPARM(12)*.

- **IPARM(15), type I:**

  WGSMP may invest significant effort during the analysis and symbolic phase in an attempt to optimize the subsequent factorization steps. While doing so, it assumes that this effort will be amortized among several factorization steps. However, if only one (or very few) factorizations are performed with the same sparsity structure, then it may be worthwhile to perform a fast analysis and reordering, even if the resulting factorization is somewhat inefficient. The input *IPARM(15)* can be used to guide *WGSMP* to apportion computational resources appropriately between the analysis and factorization steps. In *IPARM(15)*, the user should place the approximate anticipated number of factorizations that would be performed with matrices of the same structure but different values. If *IPARM(15)* is 0, which is the default, then *WGSMP* assumes a large number of factorizations per analysis step. If *IPARM(15)* $= 1$ or a small number, then *WGSMP* performs a faster analysis and reordering to minimize the overall run time, even though the factorization may run somewhat slower.

- **IPARM(16), type I:**

  IPARM(16:20) control the ordering or the generation of the fill-reducing and load-balancing permutations for the input matrix. If *IPARM(16)* $= 0$, then all default ordering options are used and speed of 3 is chosen (see below for description of speed). If *IPARM(16)* $= 1$, 2, or 3, then the options described below are used for *IPARM(17:20)* instead of the defaults. In addition, the ordering speed and quality is determined by the integer value (speed) in *IPARM(16)*. Speed $= 1$ results in the slowest but best ordering, speed $= 3$ results in fastest but worst ordering, and speed $= 2$ results in an intermediate and quality or ordering.

- **IPARM(17), type I:**

  WSMP uses graph-partitioning based ordering algorithms [4] to minimize fill during factorization. *IPARM(17)* specifies the maximum number of nodes that a subgraph must have before it is ordered by using a minimum local fill algorithm without further subpartition. The user can obtain a pure minimum local fill ordering by specifying *IPARM(17)* greater than *N*. A value of 0 in this field lets the ordering routine chose its own default. Typically, it is best to use the default, but advanced users may experiment with this parameter to find out what best suits their application. Sometimes a value larger than the default, which is between 50 and 200, may result in a faster ordering without a big compromise in quality. The default value for *IPARM(17)* is 0.

- **IPARM(18), type I:**

  The default value of 0 in *IPARM(18)* has no effect. *IPARM(18)* $= 1$ forces the ordering routine to compute a minimum local fill ordering in addition to the ordering based on recursive graph bisection. It then computes the amount of fill-in that each ordering would generate during factorization and returns the permutation corresponding to the better ordering. The use of this option increases the ordering time (in most cases the increase is not significant), but is useful when one ordering is used for multiple factorizations. Note that using this option produces the best ordering it can with the resources available to it. If graph partitioning fails due to lack of memory, it still returns the minimum local fill ordering.

- **IPARM(19), type I:**

  On input, *IPARM(19)* contains a random number seed. One can use different values of the seed to force the ordering routine to generate a different initial permutation of the graph. This is useful if one needs to generate a few different orderings of the same sparse matrix (perhaps to chose the best) without having to change the input.

- **IPARM(20), type I:**

  The input *IPARM(20)* lets the user communicate some known characteristics of the sparse matrix to *WGSMP* to aid it in choosing appropriate values of some internal parameters and to chose appropriate algorithms in various stages of ordering. If the user has no information about the type of sparse matrix or if the matrix does not fall into one of the following categories, then the default value 0 should be used. Sometimes, sparse matrices arise from finite-element graphs in which many or most vertices have more than one degree of freedom. In such graphs, there are a many small groups of nodes that share the same adjacency structure. If the sparse matrix comes from a problem like this, then a value of 2 should be used in *IPARM(20)*. This instructs *WGSMP* to construct a compressed graph before proceeding with the ordering, which then runs much faster as it runs on the smaller compressed graph rather than the original larger graph.

- **IPARM(21), type I:**

  If *IPARM(21)* $= 1$, which is the default, then *WGSMP* attempts to reorder the coefficient matrix into a block-triangular form during the analysis and reordering phase. For certain sparse systems, especially those that are highly unsymmetric in structure, this can lead to significant savings in factorization time and memory. Setting *IPARM(21)* $= 0$ suppresses block triangulation. Suppressing block triangulation may improve accuracy in rare cases by allowing all elements of the matrix to participate in partial pivoting.

- **IPARM(22), type I or M:**

  If block triangulation is attempted, then *IPARM(22)*, upon return from the analysis and reordering phase, contains the number of diagonal blocks of reasonable size that were detected by *WGSMP*. A return value of 1 indicates that reduction to block triangular form did not succeed because only one block (equivalent to the original matrix) was found.

- **IPARM(23), type O:**

  Upon return from factorization, *IPARM(23)* contains the total number of double words of memory used for storing the L and U factor matrices. Both L and U contain the diagonal, though the diagonal of L implicitly contains all ones. *WGSMP* uses relaxed supernodes to maximize the use of level-3 BLAS in factorization; i.e., it often artificially introduces explicitly stored zeros in order to obtain thick chunks of contiguous rows and columns with the same structure. This causes additional fill-in and increases the factorization operation count. The output in *IPARM(23)* includes these extra entries that are introduced to increase the size of supernodes.

- **IPARM(24), type O:**

  In *IPARM(24)*, the analysis phase returns the anticipated number of double words required to store L and U, provided that there are no row interchanges during factorization. Just like *IPARM(23)*, the output in *IPARM(24)* includes the extra factor entries that are introduced to increase the size of supernodes.

- **IPARM(25), type I:**

  IPARM(25) $= 0$, which is the default, has no effect. If *IPARM(25)* $= 1$ during iterative refinement, then the component-wise backward error is output in *DARRAY*.

- **IPARM(26), type O:**

  IPARM(26), upon return from iterative refinement, contains the number of refinement steps performed.

- **IPARM(27), type I:**

  As mentioned earlier, by default, *WGSMP* passes the coefficient matrix through a step of row permutation and scaling in order to maximize the product of the magnitudes of its diagonal entries. If a number of factorizations with matrices of the same structure but different numerical values is performed, then *WGSMP* does not re-evaluate this row permutation in each factorization step, but does so only occasionally. If *IPARM(27)* is set to 0 (which is the default), then *WGSMP* determines determines automatically when to re-evaluate the row permutation and scaling. However, if the user sets *IPARM(27)* to a positive integer, then a re-evaluation of the row permutation and scaling vectors is performed at least after every *IPARM(27)* factorization steps.

- **IPARM(28), type I:**

  By default, *WGSMP* performs partial threshold pivoting using row interchanges. However, for some matrices, this may not sufficient to prevent excessive growth along the columns. By setting *IPARM(28)* to 1, the user can instruct *WGSMP* to perform both row and column pivoting. The default value of *IPARM(28)* is 0. Note that this option should be used only when absolutely necessary, because it has the potential to slow down the factorization considerably.

- **IPARM(29), type I:**

  During factorization, *WGSMP* may end up with data structures that it allocates but does not fully use due to changes in the predicted structure of the factors due to partial pivoting. Only if *WGSMP* runs short of memory during factorization, it goes through a garbage collection step to reclaim the unused allocated space. As a result, after factorization, more memory may be tied up than the size of the factors. This situation would be harmless in most circumstances, especially is the user uses the -bmaxdata option while linking to use more virtual memory than the real memory on the machine. This is the reason why, by default, *WGSMP* does not spend time in reclaiming this memory unless absolutely needed. By setting *IPARM(29)* to 1, the user can force *WGSMP* to always return with only as much memory allocated as needed to store the factors.

- **IPARM(30:63), type R:**

  These are reserved for future use.

- **IPARM(64), type O:**

  In the event of a successful return from *WGSMP*, *IPARM(64)* is set to 0 on output. A nonzero value of *IPARM(64)* upon output indicates that *WGSMP* did not complete execution and detected an error condition. There are two types of error codes—negative and positive.

  Negative Error Codes: If an input argument error is detected, then *IPARM(64)* is set to a negative integer whose absolute value is the number of the erroneous input argument. Only minimal input argument checking is performed and a non-negative value of *IPARM(64)* does not guarantee that all input arguments have been verified to be correct. An error in the input arguments can easily go undetected and cause the program to crash or hang.

  If dynamic memory allocation by *WGSMP* fails, then *IPARM(64)* is set to -102 on return. This is one of the most common error codes encountered by the users and can often be corrected by the appropriate use of the *-bmaxdata* option while linking (see Note 2.2 for more details).

  An output value of -700 for *IPARM(64)* indicates an internal error and should be reported to *anshul@watson.ibm.com*.

  Positive Error Codes: A positive integer value of *IPARM(64)* between 1 and N on output indicates a computational error. In this case, *IPARM(64)* is the index of the first pivot that was equal to zero. A zero pivot can occur even for a non-singular matrix if the user opts for no pivoting or static pivoting. If *WGSMP* is instructed to perform threshold partial pivoting, then a zero pivot can occur for singular or nearly singular coefficient matrices. If C-style (0-based) indexing is used and *IPARM(64)* > 0, then *IPARM(64)* is 1 + the index of the bad pivot.

5.3.10 DPARM, (type I, O, M, and R): double precision parameter array

```
DOUBLE PRECISION DPARM ( 64 )
```

The entries *DPARM(29)* through *DPARM(64)* are reserved. Unlike *IPARM*, only a few of the first 25 entries of *DPARM* are used. The description of only the relevant entries of *DPARM* is given below.

- **DPARM(4), type O:**

  This is an output of step 2 (LU factorization) and contains the diagonal element of the factor with the largest magnitude.

- **DPARM(5), type O:**

  This is an output of step 2 (LU factorization) and contains the diagonal element of the factor with the smallest magnitude.

- **DPARM(6), type I:**

  DPARM(6) provides a means of performing none or fewer than *IPARM(6)* steps of iterative refinement if a satisfactory level of accuracy of the solution has been achieved. Iterative refinement is stopped if *IPARM(7)* > 0 and the backward error becomes less than *DPARM(6)*. *DPARM(6)* is not used if *IPARM(7)* = 0.

- **DPARM(7), type O:**

  If a triangular solve or iterative refinement step is performed, then *DPARM(7)* contains the backward error $\frac{\|b-A\overline{x}\|}{\|b\|}$ on output. The type of norm is determined by *IPARM(7)*. If *NRHS* > 1, then the this field contains the maximum of the backward errors amongst the *NRHS* right-hand side vectors.

- **DPARM(11), type I or M:**

  DPARM(11) is ignored if *IPARM(11)* = 0; else if *DPARM(11)* is > 0.0, then it is used as the threshold for pivoting. If *IPARM(11)* = 1 and *DPARM(11)* = 0.0 on input, then *WGSMP* chooses and appropriate pivoting threshold. Please refer to the description of *IPARM(11)* for more details. *DPARM(11)* must be non-negative.

- **DPARM(12), type I:**

  DPARM(12) is also used to provide user some control over pivoting. See the description of *IPARM(12)* for more details. *DPARM(12)* must be non-negative.

- **DPARM(22), type I:**

  DPARM(22) is used to perturb the a diagonal entry if doing so avoids a row-interchange and the perturbation option is turned on by the user by setting *IPARM(12)* $= 1$. Please refer to the description of *IPARM(12)* for more details. *DPARM(22)* must be non-negative.

- **DPARM(23), type O:**

  This contains the actual number of floating point operations performed during LU factorization. The output in *DPARM(23)* includes the extra operations that are introduced to increase the size of supernodes.

- **DPARM(24), type O:**

  This contains the number of floating point operations that the analysis phases anticipates numerical factorization to perform if no row-interchanges are performed. The output in *DPARM(24)* includes the extra operations that are introduced to increase the size of supernodes.

- **DPARM(27), type I:**

  As mentioned earlier, by default, *WGSMP* passes the coefficient matrix through a step of row permutation and scaling in order to maximize the product of the magnitudes of its diagonal entries. If a number of factorizations with matrices of the same structure but different numerical values is performed, then *WGSMP* does not re-evaluate this row permutation in each factorization step, but does so only occasionally. If *IPARM(27)* is set to 0 (which is the default), then *WGSMP* determines determines automatically when to re-evaluate the row permutation and scaling. This determination is bsed on the relative speed of factorization and the maximum matching algorithm for evaluating the row permutation, the rate of deterioration of factorization time as factorizations proceed with the old row permutation and the rate of deterioration of the condition number estimate as factorizations proceed with the old scaling vectors. By using *DPARM(27)*, the users can exert some control over how much of a deterioration in the condition number will be tolerated before re-evaluating the row permutation and scaling vectors. The default value of *DPARM(27)* is 1.0. Lowering this value lowers the tolerance to a deterioration in the condition number estimate (thus, usually prompting more frequent re-evaluation of the row permutation and the scaling vectors). Increasing *DPARM(27)* has the opposite effect.

6 Subroutines Providing a Simpler Interface

In this section, we describe a simpler interface to *WGSMP*. This interface accepts the input in both CSR and CSC formats and expects a FORTRAN-style indexing starting from 1. The shape, size, attributes, and meaning of all data structures is the same as in the calling sequence of the *WGSMP* routine described in Section 5, *unless mentioned otherwise*. Appendix B contains an example driver program that uses the simple interface. Appendix C gives the source code of many of the simple subroutines to show how they are implemented via calls to *WGSMP*. The calling sequences of these subroutines are described below:

6.1 WGCALZ (analyze, CSC input) and WGRALZ (analyze, CSR input)

WGCALZ (N, IA, JA, AVALS, NNZ, OPC, INFO)

WGRALZ (N, IA, JA, AVALS, NNZ, OPC, INFO)

These routines perform both ordering and symbolic factorization; i.e., all the preprocessing that is required prior to numerical factorization. After the completions of this preprocessing (also known as the analyze phase) any number of calls to numerical factorization and triangular solve can be made as long as the nonzero structure of the coefficient matrices does not change. The descriptions of N, *IA*, *JA*, and *AVALS* are the same as in Section 5.3. *NNZ* is an integer output containing the number of nonzeros in the potential LU factors. *OPC* is a double precision output that contains the number of floating point operations required for factorization. *INFO* is an integer output that is identical to *IPARM(64)* as described in Section 5.3.9.

6.2 WGCLUF (factor, CSC input) and WGRLUF (factor, CSR input)

WGCLUF (N, IA, JA, AVALS, THRESH, INFO)

WGRLUF (N, IA, JA, AVALS, THRESH, INFO)

These routines perform LU factorization. The descriptions of N, *IA*, *JA*, and *AVALS* are the same as in Section 5.3. *THRESH* is a double precision input that must contain the pivoting threshold, a double precision value between 0.0 and 1.0 (both inclusive). If *THRESH* is 0.0, then partial pivoting is not performed. A value of 0.05

is recommended and yields fast and accurate results for most sparse systems. *INFO* is an integer output that is identical to *IPARM(64)* as described in Section 5.3.9.

6.3 WGCSLV (solve, CSC input) and WGRSLV (solve, CSR input)

WGCSLV (N, IA, JA, AVALS, B, LDB, NRHS, NITER, BERR, INFO)

WGRSLV (N, IA, JA, AVALS, B, LDB, NRHS, NITER, BERR, INFO)

These routines solve the lower and upper triangular systems given a LU factorization and the right-hand side vector/matrix B. The descriptions of N, *IA*, *JA*, *AVALS*, B, *LDB*, *NRHS* are the same as in Section 5.3. *NITER* is an integer input by means of which the user can specify the maximum number of iterative refinement steps to be performed. If *NITER* is set to 0 on input, then iterative refinement is not performed. *BERR* is a double precision output containing the maximum relative backward error; i.e., $\frac{\|b-A\overline{x}\|_\infty}{\|b\|_\infty}$. *INFO* is an integer output that is identical to *IPARM(64)* as described in Section 5.3.9.

7 Miscellaneous Routines

In this section, we describe some optional routines available to the users for managing memory allocation and data distribution.

7.1 *WSMP_INITIALIZE ()*

This routine is used to initialize *WSMP* and *PWSMP*. Its use is optional, but if used, a call to *WSMP_INITIALIZE* must be placed right at the beginning of your program. It should be called on all nodes in the message-passing parallel mode. *WSMP* performs self initialization when the first call to any user-callable *WSMP* routine is made; however, in certain situations, as explained in Section 2.4, explicit initialization may be required.

7.2 *WSMP_CLEAR ()*

The sparse solver in *WGSMP* has an internally stored context that enables it to perform a desired task using the information from tasks performed earlier. For example, several calls to LU factorization, triangular solves, and iterative refinement can be made with different data in *AVALS* and B (but the same indices in *IA* and *JA*) after one step of symbolic factorization. The solvers are able to perform these operations because they remember the results of the last symbolic factorization. Similarly, they remember the factor for any number of solves and iterative refinement steps until a new factorization or symbolic factorization is performed to replace the previously stored information. As a result, the solver routines occupy storage to remember all the information that might be needed for a future call to perform any legal task. The user can call the routine *WSMP_CLEAR()* to free this storage if required. This routine can also be used with the simple interfaces described in Section 6. After a call to *WSMP_CLEAR()*, the solver does not remember any context and the next call, if any, must be for performing the analysis step.

WSMP_CLEAR undoes the effects of *WSMP_INITIALIZE*.

7.3 *WGFFREE ()*

Many applications perform ordering and symbolic factorizations only once for several iterations of factorization and solution. *WGSMP* allocates memory for factorization on the first call that performs factorization. This space is not released after factorization or even after subsequent triangular solves because the user can potentially make further calls for solution with the same factorization. However, the user can free this space by calling *WGFFREE ()* to use this space for tasks requiring memory allocation between factorizations. Remember, however, that this space will be reallocated in the next call to factorization and can only be temporarily reclaimed.

7.4 *WS_SORTINDICES_I (M, N, IA, JA, INFO)*

This routine can be used to sort the rows indices of each column or the column indices or each row (depending on the type of storage) of an $M \times N$ sparse matrix. The size of *IA* is $M + 1$ and the range of indices in *JA* is 0 to $N - 1$ or 1 to N. Only JA is modified upon successful completion, which is indicated by a return value of 0 in *INFO*. The descriptions of *IA* and *JA* are similar to those in Section 5.3. The description of *INFO* is similar to that of *IPARM(64)*.

7.5 *WS_SORTINDICES_V (M, N, IA, JA, AVALS, INFO)*

This routine is similar to *WS_SORTINDICES_I*, except that it also moves the values in *AVALS* according to the sorting of indices in *JA*. The descriptions of *IA*, *JA*, and *AVALS* are similar to those in Section 5.3. The description of *INFO* is similar to that of *IPARM(64)*.

7.6 *WSETMAXTHRDS (MAXTHRD)*

If the underlying hardware is an SMP platform, then *WSMP* automatically spawns threads to utilize all the available CPU's. The number of threads spawned is a function of the the number of CPU's on the given node or workstation. The routine *WSETMAXTHRDS* can be used to set the maximum number of threads that would be spawned to the input integer *MAXTHRD*.

If this routine is used, it must be called before the first call to any other *WSMP* routine.

8 Notice: Terms and Conditions for Use of *WSMP* and *PWSMP*

The information contained in this document, including the calling sequences and the terms and condition of use, are subject to change at any time without prior notice. The libraries may be used solely for educational, research, and benchmarking purposes by non-profit academic institutions and U.S. government agencies. Commercial and other organizations may make use of the libraries for benchmarking purposes only. Any other use of the libraries requires prior written permission from IBM Corporation. As unestablished research software, the libraries are provided free of charge on an "as is" basis without warranty of any kind, either expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose. IBM does not warrant that the functions contained in these libraries will meet the user's requirements or that the operation of its routines will be uninterrupted or error-free. Acceptance and use of this software constitutes the user's understanding that he/she will have no recourse to IBM for any actual or consequential damages, including, but not limited to, lost profits or savings, arising out of the use or inability to use these libraries. Even if the user informs IBM of the possibility of such damages, IBM expects the user to accept the risk of any such harm, or the user shall not attempt to use these libraries for any purpose. The libraries may not be sold, redistributed, or placed at a public site from where they can be copied by anyone other than the original requester. One may make copies of the libraries, provided that the copies are not distributed and are used under the same conditions. The use of the *WSMP* and *PWSMP* libraries is an implicit agreement by the user to these conditions.

References

[1] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18(1):140–158, January 1997.

[2] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. Technical Report RAL-TR-1999-030, Rutherford Appleton Laboratory, April 19, 1999.

[3] Anshul Gupta. WSMP: Watson sparse matrix package (Part-I: direct solution of symmetric sparse systems). Technical Report RC 21886 (98462), IBM T. J. Watson Research Center, Yorktown Heights, NY, November 16, 2000.

[4] Anshul Gupta. Graph partitioning based sparse matrix ordering algorithms for finite-element and optimization problems. In *Proceedings of the Second SIAM Conference on Sparse Matrices*, October 1996.

[5] Anshul Gupta, Mahesh Joshi, and Vipin Kumar. WSSMP: Watson symmetric sparse matrix package: The user interface. Technical Report RC 20923 (92669), IBM T. J. Watson Research Center, Yorktown Heights, NY, July 17, 1997.

[6] Anshul Gupta and Yanto Muliadi. An experimntal comparison of some direct sparse solver packages. Technical Report RC 21862 (98393), IBM T. J. Watson Research Center, Yorktown Heights, NY, October 25, 2000.

[7] Anshul Gupta and Lexing Ying. Algorithms for finding maximum matchings in bipartite graphs. Technical Report RC 21576 (97320), IBM T. J. Watson Research Center, Yorktown Heights, NY, October 19, 1999.

[8] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Supercomputing '98 Proceedings*, 1998.

[9] Markus Olschowka and Arnold Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

9 Appendix A: wgsmp_ex1.f

9.1 A serial example program showing the use of *wgsmp*

```
C------------------------------------------------------------------------------
C       Example program to show the use of the "wgsmp" routine.
C------------------------------------------------------------------------------
C.. This program can be downloaded via anonymous FTP from the following site:
C
C   ftp://ftp.cs.umn.edu/users/kumar/anshul/wgsmp_ex1.f
C
C   (C) IBM Corporation, 2000.
C
c       Acceptance and use of this program constitutes the user's understanding
c       that he/she will have no recourse to IBM for any actual or consequential
c       damages, including, but not limited to, lost profits or savings, arising
c       out of the use or inability to use this program.
C------------------------------------------------------------------------------
        program wgsmp_ex1
        implicit none

        integer n, ldb, nrhs
        integer iparm(64)
        integer ia(10)
        integer ja(32)
        double precision dparm(64)
        double precision avals(32)
        double precision b(9)
        double precision darray          ! just a placeholder in this program

        integer i
        double precision waltime, rtc

C.. Fill all arrays containing matrix data.

        data n /9/, ldb /9/, nrhs /1/

        data ia /1,5,9,12,14,17,22,24,29,33/

        data ja
     1        /1,          3,                      7,    8,
     2               2,    3,          5,                      9,
     3         1,          3,                      7,
     4                           4,                      8,
     5                                 5,    6,                9,
     6               2,          4,          6,    7,    8,
     7                           4,                7,
     8         1,          3,          5,          7,    8,
     9               2,    3,                            8,    9/
        data avals
     1      /14.d0,      -5.d0,                  -1.d0,-6.d0,
     2             14.d0,-1.d0,      -3.d0,                  -1.d0,
     3       -1.d0,      16.d0,                  -2.d0,
     4                         14.d0,                  -3.d0,
     5                               14.d0,-1.d0,            -1.d0,
     6             -2.d0,      -1.d0,      16.d0,-2.d0,-4.d0,
     7                         -1.d0,            16.d0,
     8       -3.d0,      -4.d0,      -3.d0,      -4.d0,71.d0,
     9             -1.d0,-2.d0,                        -4.d0,16.d0/

C.. Executable part of the program starts here.
```

```
      waltime = rtc()

C.. Fill 'iparm' and 'dparm' arrays with default values.

C.. As an alternative to this step, the values in 'iparm' and 'dparm' can be
c   filled with values suitable for the application either manually or by
c   using a "data" statement according to the description in the User's guide.

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0
      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      print *,'Initialization complete in time - ',rtc()-waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        stop
      end if

C.. Analysis.

      waltime = rtc()
      iparm(2) = 1
      iparm(3) = 1
      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      print *,'Analysis complete in time - ',rtc()-waltime

      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        stop
      end if

      print *,'Number of nonzeros in LU factors = ',iparm(24)
      print *,'Number of FLOPS in factorization = ',dparm(24)

C.. Factorization.

      waltime = rtc()
      iparm(2) = 2
      iparm(3) = 2
      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      waltime = rtc() - waltime
      print *,'Factorization complete in time - ',waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        stop
      end if
      print *,'Factorization MegaFlops = ',(dparm(23)*1.d-6)/waltime

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do

      waltime = rtc()
      iparm(2) = 3
      iparm(3) = 3
      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      print *,'Back substitution complete in time - ',rtc()-waltime
```

```
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        stop
      end if

C.. Iterative refinement.

      waltime = rtc()
      iparm(2) = 4
      iparm(3) = 4
      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      print *,'Iterative refinement done in time - ',rtc()-waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        stop
      end if

      print *,'Maximum relative error = ',dparm(7)
      print *,'The solution of the system is as follows:'
      do i = 1, n
        print *,i,' : ',b(i)
      end do

      stop
      end
```

9.2 A script showing the execution of program *wgsmp_ex1*

```
% xlf_r -O3 -qstrict -lwsmp -lpthreads -lm_r wgsmp_ex1.f
** wgsmp_ex1   === End of Compilation 1 ===
1501-510  Compilation successful for file wgsmp_ex1.f.
% setenv SPINLOOPTIME 500
% setenv YIELDLOOPTIME 500
% a.out
 Initialization complete in time -  0.107921361923217773E-01
 Analysis complete in time -  0.934123992919921875E-03
 Number of nonzeros in LU factors =  45
 Number of FLOPS in factorization =  176.000000000000000
 Factorization complete in time -  0.480771064758300781E-03
 Factorization MegaFlops =  0.366078589377006047
 Back substitution complete in time -  0.469088554382324219E-03
 Iterative refinement done in time -  0.257492065429687500E-04
 Maximum relative error =  0.437150315946155388E-15
 The solution of the system is as follows:
 1  :  0.117399109178044914
 2  :  0.101283273917971053
 3  :  0.782595739613466124E-01
 4  :  0.780325936280052007E-01
 5  :  0.844629723197335147E-01
 6  :  0.961642685452215457E-01
 7  :  0.673770371017503172E-01
 8  :  0.308187702640242700E-01
 9  :  0.863173439310475910E-01
%
```

10 Appendix B: wgsmp_ex2.f

10.1 A serial example program showing the use of the simple interface

```
C-------------------------------------------------------------------------------
C       Example program to show the use of the simple interface for
c       solving general sparse systems in the WSMP library.
C-------------------------------------------------------------------------------
C.. This program can be downloaded via anonymous FTP from the following site:
C
C   ftp://ftp.cs.umn.edu/users/kumar/anshul/wgsmp_ex2.f
C
C   (C) IBM Corporation, 2000.
C
c       Acceptance and use of this program constitutes the user's understanding
c       that he/she will have no recourse to IBM for any actual or consequential
c       damages, including, but not limited to, lost profits or savings, arising
c       out of the use or inability to use this program.
C-------------------------------------------------------------------------------
        program wgsmp_ex2
        implicit none

        integer n, ldb, nrhs, niter, nnz, info
        integer ia(10)
        integer ja(32)
        double precision avals(32)
        double precision b(9)
        double precision berr, thresh, opc
        double precision darray          ! just a placeholder in this program

        integer i
        double precision waltime, rtc

C.. Fill all arrays containing matrix data.

        data n /9/, ldb /9/, nrhs /1/, niter /3/
        data thresh /5.d-2/
        data ia /1,5,9,12,14,17,22,24,29,33/

        data ja
     1        /1,          3,                      7,    8,
     2               2,    3,          5,                      9,
     3         1,          3,                      7,
     4                           4,                      8,
     5                                 5,    6,                9,
     6               2,          4,          6,    7,    8,
     7                           4,                7,
     8         1,          3,          5,          7,    8,
     9               2,    3,                            8,    9/
        data avals
     1      /14.d0,      -5.d0,                  -1.d0,-6.d0,
     2             14.d0,-1.d0,      -3.d0,                  -1.d0,
     3       -1.d0,      16.d0,                  -2.d0,
     4                         14.d0,                  -3.d0,
     5                               14.d0,-1.d0,            -1.d0,
     6             -2.d0,      -1.d0,      16.d0,-2.d0,-4.d0,
     7                         -1.d0,            16.d0,
     8       -3.d0,      -4.d0,      -3.d0,      -4.d0,71.d0,
     9             -1.d0,-2.d0,                        -4.d0,16.d0/

C.. Analysis.
```

```
      waltime = rtc()
      call wgralz (n, ia, ja, avals, nnz, opc, info)

      print *,'Analysis complete in time - ',rtc()-waltime

      if (info .ne. 0) then
        print *,'The following ERROR was detected: ',info
        stop
      end if

      print *,'Number of nonzeros in LU factors = ',nnz
      print *,'Number of FLOPS in factorization = ',opc

C.. Factorization.

      waltime = rtc()
      call wgrluf (n, ia, ja, avals, thresh, info)

      waltime = rtc() - waltime
      print *,'Factorization complete in time - ',waltime
      if (info .ne. 0) then
        print *,'The following ERROR was detected: ',info
        stop
      end if
      print *,'Factorization MegaFlops = ',(opc*1.d-6)/waltime

C.. Back substitution with iterative refinement.

      do i = 1, n
        b(i) = 1.d0
      end do

      waltime = rtc()
      call wgrslv (n, ia, ja, avals, b, ldb, nrhs, niter, berr, info)

      print *,'Back substitution complete in time - ',rtc()-waltime
      if (info .ne. 0) then
        print *,'The following ERROR was detected: ',info
        stop
      end if

      print *,'Maximum relative error = ',berr
      print *,'The solution of the system is as follows:'
      do i = 1, n
        print *,i,' : ',b(i)
      end do

      stop
      end
```

10.2 A script showing the execution of program *wgsmp_ex2*

```
% xlf_r -O3 -qstrict -lwsmp -lpthreads -lm_r wgsmp_ex2.f
** wgsmp_ex2   === End of Compilation 1 ===
1501-510  Compilation successful for file wgsmp_ex2.f.
% setenv SPINLOOPTIME 500
% setenv YIELDLOOPTIME 500
% a.out
 Analysis complete in time -  0.161006450653076172E-01
 Number of nonzeros in LU factors =  45
 Number of FLOPS in factorization =  176.000000000000000
```

```
 Factorization complete in time -  0.430464744567871094E-03
 Factorization MegaFlops =  0.408860428690113520
 Back substitution complete in time -  0.118076801300048828E-02
 Maximum relative error =  0.437150315946155388E-15
 The solution of the system is as follows:
 1  :  0.117399109178044914
 2  :  0.101283273917971053
 3  :  0.782595739613466124E-01
 4  :  0.780325936280052007E-01
 5  :  0.844629723197335147E-01
 6  :  0.961642685452215457E-01
 7  :  0.673770371017503172E-01
 8  :  0.308187702640242700E-01
 9  :  0.863173439310475910E-01
%
```

11 Appendix C: The source code of the simple interface

```
C-----------------------------------------------------------------------------
C   (C) IBM Corporation, 2000.
C
c   Acceptance and use of these programs constitutes the user's understanding
c   that he/she will have no recourse to IBM for any actual or consequential
c   damages, including, but not limited to, lost profits or savings, arising
c   out of the use or inability to use this program.
C-----------------------------------------------------------------------------
      subroutine WGRALZ ( N, IA, JA, AVALS, NNZ, OPC, INFO )
      integer iparm(64)
      double precision opc, dparm(64)
      integer ldb, nrhs              ! just place holders
      double precision b, darray     ! just place holders

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0

C       Initialize IPARM and DPARM with default values

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      iparm(2) = 1                   ! first job is analysis
      iparm(3) = 1                   ! last job is analysis

C       Perform ordering and symbolic factorization

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      nnz = iparm(24)
      opc = dparm(24)
      info = iparm(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine WGCALZ ( N, IA, JA, AVALS, NNZ, OPC, INFO )
      integer iparm(64)
      double precision opc, dparm(64)
      integer ldb, nrhs              ! just place holders
      double precision b, darray     ! just place holders

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0

C       Initialize IPARM and DPARM with default values

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      iparm(2) = 1                   ! first job is analysis
      iparm(3) = 1                   ! last job is analysis
      iparm(4) = 1                   ! input matrix is in CSC format

C       Perform ordering and symbolic factorization

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      nnz = iparm(24)
      opc = dparm(24)
      info = iparm(64)
```

```
      return
      end
C-----------------------------------------------------------------------------
      subroutine WGRLUF ( N, IA, JA, AVALS, THRESH, INFO )
      integer iparm(64)
      double precision thresh, dparm(64)
      integer ldb, nrhs              ! just place holders
      double precision b, darray     ! just place holders

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0

C       Initialize IPARM and DPARM with default values

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      iparm(2) = 2                   ! first job is LU factorization
      iparm(3) = 2                   ! last job is LU factorization
      dparm(11) = thresh

C       Perform LU factorization

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      info = iparm(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine WGCLUF ( N, IA, JA, AVALS, THRESH, INFO )
      integer iparm(64)
      double precision thresh, dparm(64)
      integer ldb, nrhs              ! just place holders
      double precision b, darray     ! just place holders

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0

C       Initialize IPARM and DPARM with default values

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      iparm(2) = 2                   ! first job is LU factorization
      iparm(3) = 2                   ! last job is LU factorization
      iparm(4) = 1                   ! input matrix is in CSC format
      dparm(11) = thresh

C       Perform LU factorization

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      info = iparm(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine WGRSLV ( N,IA,JA,AVALS,B,LDB,NRHS,NITER,BERR,INFO )
      integer iparm(64)
      double precision berr, dparm(64)
      double precision darray        ! just a place holder

      iparm(1) = 0
      iparm(2) = 0
```

```
      iparm(3) = 0

C       Initialize IPARM and DPARM with default values

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      iparm(2) = 3                    ! first job is triangular solve
      iparm(3) = 4                    ! last job is iterative refinement
      iparm(6) = niter

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      berr = dparm(7)
      info = iparm(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine WGCSLV ( N,IA,JA,AVALS,B,LDB,NRHS,NITER,BERR,INFO )
      integer iparm(64)
      double precision berr, dparm(64)
      double precision darray        ! just a place holder

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0

C       Initialize IPARM and DPARM with default values

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      iparm(2) = 3                    ! first job is triangular solve
      iparm(3) = 4                    ! last job is iterative refinement
      iparm(4) = 1                    ! input matrix is in CSC format
      iparm(6) = niter

      call wgsmp (n, ia, ja, avals, b, ldb, nrhs, darray, iparm, dparm)

      berr = dparm(7)
      info = iparm(64)
      return
      end
C-----------------------------------------------------------------------------
```