RC 20923 (92669) July 17, 1997
Computer Science/Mathematics

IBM Research Report

WSSMP: Watson Symmetric Sparse Matrix Package

Users Manual: Version 2.3β

Anshul Gupta

IBM Research Division
T.J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
anshul@watson.ibm.com

Mahesh Joshi and **Vipin Kumar**

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
{*mjoshi* and *kumar*}*@cs.umn.edu*

Research Division
Almaden · Austin · China · Haifa · Tokyo · Watson · Zurich

WSSMP: Watson Symmetric Sparse Matrix Package

Users Manual: Version 2.3β^*

Anshul Gupta

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

anshul@watson.ibm.com

Mahesh Joshi and **Vipin kumar**

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

{*mjoshi* and *kumar*}*@cs.umn.edu*

IBM Research Report RC 20923 (92669)

July 17, 1997

*Since this document describes software that might be upgraded from time to time, *WSSMP* users and the readers of this report are encouraged to request the latest version of the manual via e-mail from the first author. The software and the documentation was last updated on July 31, 1997

Contents

1 Introduction

The Watson Symmetric Sparse Matrix Package, *WSSMP*, is a robust high performance software package for solving large sparse symmetric positive definite systems of linear equations on the IBM RS6000 work-stations and IBM Scalable Parallel (SP) systems. It uses a modified version of the multifrontal algorithm [8] for sparse Cholesky factorization and a highly scalable parallel sparse Cholesky factorization algorithm [6, 3]. The package also uses scalable parallel sparse triangular solvers [7] and a parallel version of the previously released package *WGPP* [4, 5] for computing fill-reducing orderings. The current release of *WSSMP* can perform any or all of the following tasks: *ordering, symbolic factorization, Cholesky factorization, triangular solves,* and *iterative refinement.* An SMP version will be released in the future. The next version is also expected to perform LDL^T factorization, in addition to LL^T factorization, which is the only factorization option in the current version.

The functionality and the calling sequences of the serial and the parallel versions are almost the same. The users manual is organized accordingly and the descriptions of most parameters for both versions is included in the description of the serial version. The serial version supports certain features that the current parallel version does not. Such features, options, or data structures supported exclusively by the serial version will be annotated by a superscript ^{S} in this document. Similarly, items relevant only to the parallel version appear with a superscript ^{P}. Section 2 lists the various libraries that are available and describes how to obtain and link the libraries. Section 3 gives a detailed description of the main serial routine that provides an advanced single routine interface to the entire software. This section also describes the input data structures for the serial case. In this section, the differences from the parallel version are noted, wherever applicable. Section 4 describes user callable routines that provide a simpler interface to the serial solver, but omit some of the advanced features. Section 5 describes the two parallel modes in which the the solver can be used, describes the input data structures for the parallel solution, and reminds users of the differences between the serial and the parallel versions, wherever applicable. This section does not repeat the information contained in Section 3 because the parallel user-interface is very similar to the serial one. Section 6 is the parallel analog of Section 4 and describes user callable routines that provide a simpler interface to the parallel solver. 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 known problems associated with the use of this package. Section 9 contains the terms and conditions that every user of the package must adhere to.

2 Obtaining and Linking the Libraries

The serial *WSSMP* library is currently available for the following platforms:

- IBM RS6000 POWER-2 (XLF 3.xx and higher), code WSSMPP2.
- IBM RS6000 POWER PC (XLF 3.xx and higher), code WSSMPPC.
- Entire IBM RS6000 family (XLF 3.xx and higher), code WSSMP.

The following two versions of the parallel library are available:

- IBM RS6000 POWER-2 (XLF 3.2.3.xx and higher), code PWSSMPP2.

- Entire IBM RS6000 family (XLF 3.2.3.xx and higher), code PWSSMP.

For using the serial version on an RS6000, only the appropriate *WSSMP* library needs to be linked. In order to use the parallel version on an SP machine, both *WSSMP* and *PWSSMP* libraries must be linked.

The libraries can be obtained by sending e-mail to anshul@watson.ibm.com with the subject line containing the request code followed by one or more blanks followed by the e-mail address at which the appropriate library and the manual will be e-mailed. The body of the message must contain the requester's name, affiliation, and the intended use of the software. A user must read Section 9 of this manual before using *WSSMP*. The manual 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. It may be required to run *ranlib* on the decoded files.

Note 2.1 *If the libraries are linked with C code and a C compiler is being used for linking, then the Fortran library must be linked in addition to the* WSSMP *libraries by -lxlf90 option. In addition, the -L option may be required to indicate the paths to the various libraries. Otherwise, the C code may be compiled with the -c option and a Fortran compiler should be used for linking and generating the final executable.*

2.1 Using serial WSSMP with OSL

The users of the IBM Optimization Subroutine Library (*OSL*) (Version 1.2) on an RS6000 can benefit from serial *WSSMP* by simply linking this library before *OSL* while compiling their programs. For example, on a POWER-2 based IBM RS6000 with XLF Version 3,

```
xlf -O3 -qstrict -o foo.out foo.f -lwssmpp2 -losl -lm
```

will automatically replace the sparse matrix ordering and factorization routines of *OSL* by the corresponding *WSSMP* routines (called automatically with appropriate arguments). C users will need to use one of the following ways of using *WSSMP*:

```
xlc -O3 -o foo.out foo.c -lwssmpp2 -lxlf90 -losl -lm
```

or

```
xlc -O3 -c foo.c
xlf -o foo.out foo.o -lwssmpp2 -losl -lm
```

In addition, the -L option may be required to indicate the paths to the various libraries.

3 The Primary Serial Subroutine: *WSSMP*

This section describes the use of the *WSSMP* subroutine and its calling sequences in detail. There are five basic tasks that *WSSMP* is capable of performing, namely, *ordering*, *symbolic factorization*, *Cholesky factorization*, *forward and backward solve*, and *iterative refinement* (see Note 3.5). 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 3.2). In addition, a call to *WSSMP* 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 3.2.14 for more details. The ordering is performed using the Watson Graph Partitioning (and sparse matrix ordering) Package [5]. The user can skip *WSSMP*'s ordering step and use the output (vectors *PERM* and *INVP*) generated from any ordering code, including that from the *WMMRB, WN1ORD,* and *WKKTORD* routines of *WGPP* called directly. The ordering routines of *WGPP* are included in the *WSSMP* package.

In addition to the advanced interface that the *WSSMP* library provides via the single subroutine *WSSMP*, there are a number of other subroutines that provide a simpler interface. These subroutines are described in detail in Section 4.

Note 3.1 *Calling the* WMMRB, WN1ORD, *or* WKKTORD *routines of* WGPP *directly is faster when a call to* WSSMP *is being used only to generate the ordering. However, if* WSSMP *is being called to perform multiple functions in the same call and ordering is one of them, then using* WSSMP *for ordering is usually faster than calling the* WGPP *routines separately.*

Note 3.2 *The original code for* WSSMP *is in Fortran and expects the parameters to be passed by reference. Therefore, when calling* WSSMP *from a C program, the addresses of the parameters described in Section 3.2 must be passed explicitly.*

3.1 Types of matrices accepted and their input format

The *WSSMP* routine currently works only for symmetric positive definite matrices. If the matrix is negative definite, then it is the user's responsibility to scale the matrix. It can also handle semi-definite matrices via appropriate use of the options provided by *IPARM(11:13)*, and *DPARM(11,12,21,22)*. Please refer to Section 3 for details. All floating point values must be 8-byte real numbers and all integers must be 4 bytes long.

Since the input matrix is symmetric, only a triangular part is accepted a input. Currently, two input formats are supported. In the first format, the input is the upper triangular part, including the diagonal, in compressed sparse row (CSR) or the lower triangular part in the compressed sparse column (CSC) format. The second format is modified compressed sparse row/column (MSR or MSC). The performance of *WSSMP* in the symbolic phases of the computation slightly better with the CSR/CSC format than with the MSR/MSC format. The serial version of the package supports an additional input format which is similar to the CSR format, but is somewhat more flexible in that the column (row) indices of row (column) $i+1$ don't necessarily have to follow those of row (column) i. We will call the third format the WSSMP format because this is the one used internally by the package. Figure 1 illustrates all three input formats; they are also explained briefly in Sections 3.2.2, 3.2.3, 3.2.4, and 3.2.5.

WSSMP supports both C-style indexing starting from 0 and Fortran-style indexing starting from 1. C-style (0-based) indexing is preferred and is used in the internal routines. Fortran-style (1-based) numbering forces a conversion in the beginning and a reconversion at the end. Please refer to the description of *IPARM(5)* in Section 3.2.14.

3.2 Calling sequence of the WSSMP subroutine

Please refer to Note 3.2 before reading this section. There are five types of arguments, namely input (type **I**), output (type **O**), modifiable (type **M**), temporary (type **T**), and reserved (type **R**). The input arguments are read by the *WSSMP* 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 *WSSMP* and modified to return some information, the temporary arguments are not read but their contents are overwritten by unpredictable values during execution, and the reserve

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

A 9 X 9 symmetric sparse matrix.

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

	CSR Format			MSR Format				WSSMP Format^S			
K	IA(K)	JA(K)	AVALS(K)	IA(K)	JA(K)	AVALS(K)	DIAG(K)	IA(1,K)	IA(2,k)	JA(K)	AVALS(K)
1	1	1	14.0	1	3	-1.0	14.0	1	4	1	14.0
2	5	3	-1.0	4	7	-1.0	14.0	5	4	3	-1.0
3	9	7	-1.0	7	8	-3.0	16.0	9	4	7	-1.0
4	13	8	-3.0	10	3	-1.0	14.0	13	4	8	-3.0
5	17	2	14.0	13	8	-3.0	14.0	17	4	2	14.0
6	21	3	-1.0	16	9	-1.0	16.0	21	4	3	-1.0
7	25	8	-3.0	18	7	-2.0	16.0	25	2	8	-3.0
8	27	9	-1.0	19	8	-4.0	71.0	27	2	9	-1.0
9	29	3	16.0	20	9	-2.0	16.0	29	1	3	16.0
10	30	7	-2.0	20	6	-1.0				7	-2.0
11		8	-4.0		7	-1.0				8	-4.0
12		9	-2.0		8	-3.0				9	-2.0
13		4	14.0		6	-1.0				4	14.0
14		6	-1.0		8	-3.0				6	-1.0
15		7	-1.0		9	-1.0				7	-1.0
16		8	-3.0		7	-2.0				8	-3.0
17		5	14.0		8	-4.0				5	14.0
18		6	-1.0		9	-2.0				6	-1.0
19		8	-3.0		8	-4.0				8	-3.0
20		9	-1.0		9	-4.0				9	-1.0
21		6	16.0							6	16.0
22		7	-2.0							7	-2.0
23		8	-4.0							8	-4.0
24		9	-2.0							9	-2.0
25		7	16.0							7	16.0
26		8	-4.0							8	-4.0
27		8	71.0							8	71.0
28		9	-4.0							9	-4.0
29		9	16.0							9	16.0

Figure 1: Illustration of the three input formats for the serial version.

arguments are just like temporary arguments which 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 the sparse symmetric positive definite 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 *WSSMP* and overwrites b when *WSSMP* is called to compute the solution of the system.

Note 3.3 *Recall that* WSSMP *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* WSSMP.

The calling sequence and description of the parameters of *WSSMP* is as follows:

WSSMP (N, IA, JA, AVALS, DIAG, PERM, INVP, B, LDB, NRHS, AUX, NAUX, MRP, IPARM, DPARM)

3.2.1 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 $Ax = b$.

3.2.2 IA, (type I or M): row pointers

- **Case 1:** If input format is CSR or MSR; i.e., *IPARM(4)* $= 0$ or 1 or 8 or 9 :

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

  IA is an integer array of size one greater than N. *IA(I)* points to the first column index of row I in the array *JA*. Refer to Figure 1 and description of *IPARM(4)* in Section 3.2.14 for more details.

- **Case 2^S:** If input format is WSSMP internal; i.e., *IPARM(4)* $= 2$ or 10 :

  ```
  INTEGER IA (2, N)
  ```

  Case 2 is supported only in the serial version. *IA* is a two-dimensional integer array. *IA(1,I)* points to the first column index of row I (or first row index of column I) in the array *JA*. *IA(2,I)* stores the number of nonzero entries in the I-th row in the upper-triangular portion of the matrix A, which is the same as the number of entries in its I-th column of the lower triangular portion. Refer to Figure 1 and Section 3.2.14 for more details.

  If *IPARM(8)* $= 2$, then the matrix is permuted on output. Please refer to the description of *IPARM(8)* and Note 3.6 for more details.

3.2.3 JA, (type I or M): column indices

```
INTEGER JA ( * )
```

The integer array *JA* contain the column (row) indices of the upper (lower) triangular part of the symmetric sparse matrix *A*. The indices of a column (row) are stored in consecutive locations. In addition, if *IPARM(8)* = 2; i.e., the matrix is already permuted and the user does not want *WSSMP* to permute the matrix, then these consecutive column (row) indices of a row (column) *must* be sorted in increasing order upon input. If CSR or *WSSMP* internal input format is used, then the size of array *JA* is the total number of nonzeros in a triangular portion of the symmetric matrix *A* (including the diagonal). If the MSR input format is used, then the size of *JA* is the number of nonzeros in a strictly triangular portion of *A* excluding the diagonal.

If *IPARM(8)* = 2, then the matrix is permuted on output. Please refer to the description of *IPARM(8)* and Note 3.6 for more details.

3.2.4 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. The input *AVALS* may be modified if scaling is opted for; see the description of *IPARM(10)* for more details.

If *IPARM(8)* = 2, then the matrix is permuted on output. Please refer to the description of *IPARM(8)* and Note 3.6 for more details.

3.2.5 DIAG, (type I): diagonal values

```
DOUBLE PRECISION DIAG ( N )
```

If the MSR input format is used, then *DIAG(I)* contains the *I*-th diagonal element of the coefficient matrix. If CSR or WSSMP internal formats are used for input, then *DIAG* is not referenced and need not be a double precision array of size *N*; it can simply be a placeholder (e.g., a *NULL* can be passed in C). The input *DIAG* is not modified even if scaling is opted for because scaling does not modify the user input values if MSR format is used; instead it modifies an internal copy. See the description of *IPARM(10)* for more details.

If *IPARM(8)* = 2, then the matrix is permuted on output. Please refer to the description of *IPARM(8)* for more details.

3.2.6 PERM, (type I or O): permutation vector

```
INTEGER PERM ( N )
```

PERM is the permutation vector, as defined in Sparspak [1]. If J = *PERM(I)*, then the J-th row and column of the original matrix become the I-th row and column in the permuted matrix to be factored. If ordering is one of the steps that the call to *WSSMP* is made to perform, the *PERM* is an output parameter. If *WSSMP* is called to perform a task or a set of tasks other than ordering, then *PERM* is an input parameter if either the matrix *A* of the right-hand side *B* is not permuted already. If both *A* and *B* are already permuted when passed in to *WSSMP*, then *PERM* is not referenced. See Section 3.2.14 for more details.

3.2.7 INVP, (type I or O): inverse permutation vector

```
INTEGER INVP ( N )
```

INVP is the inverse permutation vector, as defined in Sparspak [1]. If $J = INVP(I)$, then the I-th row and column of the original matrix become the J-th row and column in the permuted matrix to be factored. If ordering is one of the steps that the call to *WSSMP* is made to perform, the *INVP* is an output parameter. If *WSSMP* is called to perform a task or a set of tasks other than ordering, then *INVP* is an input parameter if either the matrix A of the right-hand side B is not permuted already. If both A and B are already permuted when passed in to *WSSMP*, then *INVP* is not referenced. See Section 3.2.14 for more details.

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

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

The $N\times$ *NRHS* dense matrix B 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 if a call to *WSSMP* includes solution or iterative refinement.

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

```
INTEGER LDB
```

LDB is the leading dimension of the right-hand side matrix if $NRHS > 1$. When used, *LDB* must be greater than or equal to N.

3.2.10 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.

3.2.11 AUX, (type O, I, or T): auxiliary storage

```
DOUBLE PRECISION AUX ( NAUX )
```

The user can pass the storage required for factorization to *WSSMP* via the optional double precision array *AUX*. If *AUX* is used by *WSSMP* during factorization, then it must be passed in unmodified for triangular solutions and iterative refinement. Whether the user supplied storage *AUX* is used by *WSSMP* or not depends on the value of the input parameter *NAUX*. If $NAUX = 0$, then *WSSMP* allocates space dynamically, otherwise if $NAUX > 0$, then the *AUX* array is used. If $NAUX > 0$, then *AUX* is also used as as integer working storage of size $r \times NAUX$ for the ordering routine *WMMRB*, where r is the ratio of the size of a double precision number to the size of an integer on the hardware platform. If *AUX* is used by *WSSMP*, then it is of type T for ordering, unused for symbolic factorization, of type O for factorization, type I for triangular solves and iterative refinement.

Note 3.4 *If* AUX *is used in a call to* WSSMP *for Cholesky factorization, then it must be passed unaltered in subsequent calls to* WSSMP *for triangular solves and iterative refinement with the same matrix.*

3.2.12 NAUX, (type I): size of user supplied auxiliary storage

```
INTEGER NAUX
```

NAUX is the number of double words in *AUX*. If *NAUX* is zero, then *WSSMP* allocates space dynamically for factorization. If *NAUX* is a positive integer, then *WSSMP* attempts to use the first *NAUX* double words of the array *AUX* for factorization. If that is not sufficient, then *WSSMP* returns without completion. *IPARM(64)* returns the status of the call and tells the user whether the *WSSMP* call was successful or not. Whether *NAUX* is 0, or insufficient, or more than sufficient, *IPARM(23)* returns the actual size of *AUX* required. *IPARM(23)* is filled with the correct value of *NAUX* after symbolic factorization and the user has a chance to inspect this value before calling *WSSMP* for Cholesky factorization. See Section 3.2.14 for more details.

3.2.13 MRP, (type M or O): pivot values or pivot info

- **Case 1^S:** If *IPARM(4)* ≥ 8 and *IPARM(11)* < 2 :

  ```
  DOUBLE PRECISION MRP ( N )
  ```

  Case 1 is supported only in the serial version. In this case, the double precision vector *MRP* contains the principal diagonal of the lower triangular Cholesky factor of A upon return from *WSSMP* if factorization is performed.

- **Case 2:** If *IPARM(11)* ≥ 2 :

  ```
  INTEGER MRP ( N )
  ```

  This case overrides case 1; i.e., if *IPARM(11)* ≥ 2 and *IPARM(4)* ≥ 8, then *IPARM(4)* is set to *IPARM(4)* - 8. In this case, on return from *WSSMP* factorization, *MRP(I)* is set to *IPARM(13)* if the I-th pivot was less than or equal to *DPARM(11)* during factorization. Otherwise, *MRP(I)* is not touched.

3.2.14 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 *WSSMP* and to return some useful information about the execution of call to *WSSMP*. If *IPARM(1)* is 0, then *WSSMP* uses default values of *IPARM(4)* through *IPARM(64)* and *DPARM*. *IPARM(1)* through *IPARM(3)* are mandatory inputs, which must always be supplied by the user. If *IPARM(1)* is 1, then *WSSMP* uses the user supplied entries in the arrays *IPARM* and *DPARM*. The default initial values of *IPARM* and *DPARM* are shown in Table 1. Note that some of the entries in *IPARM* and *DPARM* are of type M or O. If *IPARM(1)* is 0, then the default values of Table 1 are used before *WSSMP* performs the user designated task(s) it has been called to perform. The contents of *IPARM* and *DPARM* at the end of the execution, depending on the task(s) performed by *WSSMP*, are the output or modified values corresponding to the default input values. It is possible for a user to call *WSSMP* 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 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			DPARM		
Index	Default input	Description	Type	Default input	Description	Type
1	mandatory I/P	default/user defined	M	-	unused	-
2	mandatory I/P	starting task	M	-	matrix norm^S	O
3	mandatory I/P	last task	I	-	mat. inv. norm^S	O
4	0	I/P format	I	-	max. $L_{i,i}^2$	O
5	1	numbering style	I	-	min. $L_{i,i}^2$	O
6	1	no. of iter. ref.	M	1.0E-12	rel. err. lim.	I
7	0	residual norm type	I	-	relative err.	O
8	0	mat. permut. option	M	-	unused	-
9	0	RHS permut. option	M	-	unused	-
10^S	0	scaling option	M	1.0E-02	scaling range	I
11	0	bad pivot handling	I	1.0E-60	bad piv. threshold	I
12	0	small piv. handling	I	1.0E-30	small piv. thresh.	M
13	0	bad pivot flag	I	-	unused	-
14	0	AVALS reuse optn.	I	-	unused	-
15	0	N1	I	-	unused	-
16	1	WMMRB option 1	I	-	unused	-
17	0	WMMRB option 2	I	-	unused	-
18	1	WMMRB option 3	I	-	unused	-
19	0	WMMRB option 4	I	-	unused	-
20	7	WMMRB option 5	I	-	unused	-
21	-	bad pivot count	O	1.0E200	bad pivot subst.	I
22	-	small pivot count	O	1.0E-15	small piv. subst.	I
23	-	minimum NAUX	O	-	factorization ops.	O
24	-	factor nonzeros	O	-	tri. solve ops.	O
25^S	0	cond. no. option	I	-	unused	-
26^P	0	block size	M	-	unused	-
27^P	0	balancing option	I	-	unused	-
28^P	0	tri. solve blk.	I	-	unused	-
29-63	-	reserved	R	-	reserved	R
64	-	info O/P	O	-	first bad pivot	O

Table 1: The default initial values of the various entries in *IPARM* and *DPARM* arrays. A '-' indicates that the value is not read by *WSSMP*.

- **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 *WSSMP* before further computation and *IPARM(1)* itself is set to 1. If *IPARM(1)* is 1 on input, then *WSSMP* 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 *WSSMP*, if any. This is to facilitate users to restart processing on a problem from where the last call to *WSSMP* left it. If *WSSMP* performs no task, then, on output, *IPARM(2)* is set to *max(IPARM(2),IPARM(3)+ 1)*. *WSSMP* can perform any set of consecutive tasks from the following list:

Task # 1:	Ordering
Task # 2:	Symbolic Factorization
Task # 3:	Cholesky Factorization
Task # 4:	Forward and Backward Elimination
Task # 5:	Iterative Refinement

Note 3.5 WSSMP *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,* WSSMP *can work on a system in increasing order of task numbers. If a call to* WSSMP *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* WSSMP *in a previous call, then all or some of 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 *WSSMP*. In a call to *WSSMP*, 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 1–5, then no task is performed. This can be used to fill *IPARM* and *DPARM* with default values; e.g., by calling *WSSMP* 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 and instructs *WSSMP* whether or not to output the diagonal of the Cholesky factor of the coefficient matrix. *IPARM(4)* = 0 denotes CSR, *IPARM(4)* = 1 denotes MSR, and *IPARM(4)* = 2 denotes *WSSMP* internal format. All three formats are illustrated in Figure 1.

  WSSMP does not give user the access to the Cholesky factor of the coefficient matrix, which is stored in a non-standard format either in the user supplied storage area AUX, or in internally allocated data structures. However, occasionally the user might want to inspect the diagonal values of the factor to analyze the numerical accuracy of the factorization process. If 8 is added to the above mentioned values of *IPARM(4)*; i.e., if 8, 9, or 10 is used instead of 0, 1, and 2, then the parameter *MRP* is treated as a double precision array of size N and the diagonal of the Cholesky factor is copied into *MRP* (provided that task number 3 is performed during that call to *WSSMP*. In terms of interpreting the input format of A, an 8, 9, or a 10 in *IPARM(4)* is treated exactly as a 0, 1, or a 2, respectively.

  Note that only the first two input formats are supported in the parallel version (see Figure 2. In addition, the option of having the diagonal returned in *MRP* is not yet available in the parallel version. Therefore, 0 and 1 are the only valid inputs in *IPARM(4)* in the parallel version.

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

  On input, *IPARM(6)* should be set to the maximum number of steps of iterative refinement to be performed; on output, *IPARM(6)* contains the actual number of iterative 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 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 relative 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 relative error given by $||b - A\overline{x}||/||b||$ 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. If *IPARM(7)* = 1 or 5, then 1-norms are used in computing the relative 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 relative error given by $||b - A\overline{x}||/||b||$ at the end of the last iterative refinement step is placed in *DPARM(7)*. Even if iterative refinement is not performed (i.e. if *IPARM(6)* = 0 or if *IPARM(3)* < 5) the value of relative error is computed and placed in *DPARM(7)* if *IPARM(7)* = 1, 2, 3, 5, 6, or 7.

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

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

  On input, *IPARM(8)* = 0 means that the matrix A is not permuted and its permutation given by *PERM* and *INVP* must be used in symbolic factorization, Cholesky factorization, solution, and/or iterative refinement.

  IPARM(8) = 1 means that the matrix is permuted already and it must be used as is by *WSSMP*.

  IPARM(8) = 2 means that the matrix is not permuted and instructs *WSSMP* to permute the matrix and overwrite the data structures (*IA, JA, AVALS,* and *DIAG*) with the contents of the permuted matrix. In addition, if *IPARM(8)* = 2 on input, it is set to 1 on output. If *IPARM(8)* = 1 on input and ordering is one of the steps being performed, then *IPARM(8)* is treated as if it was equal to 2 on input. The input matrix is overwritten by the permuted matrix only if symbolic and/or Cholesky factorization is performed. Moreover, if symbolic factorization is performed and Cholesky factorization is not performed in a call to *WSSMP*, then the array *AVALS* is not permuted and overwritten. This is because *AVALS* may not have been available and/or may not have been to supplied to *WSSMP* in that call. Thus it is valid for *IPARM(8)* to have a value of 0 or 2 during symbolic factorization and to use *IPARM(8)* = 1 during Cholesky factorization. This can be useful because often it is cheaper to generate the matrix in permuted form than to compute the permutation of an existing matrix. Having *WSSMP* do the permutation can sometimes be expensive relative to the factorization cost, especially while using the parallel version.

Note 3.6 *If* IPARM(8) $=$ *1; i.e., the user is supplying a pre-permuted matrix, then the indices of a particular row/column in the array* JA *must be sorted in* increasing *order. If the matrix is not pre-permuted; i.e.,* IPARM(8) *is 0 or 2, then the indices of a particular row/column in* JA *can be in any order.*

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

  On input, *IPARM(9)* $= 0$ means that the right-hand side B is not permuted and its permutation given by *PERM* and *INVP* must be used by *WSSMP*. *IPARM(9)* $= 1$ means that B is permuted already and must be used as it is by *WSSMP*. *IPARM(9)* $= 2$ means that the B not permuted and instructs *WSSMP* to overwrite B with its permuted version. In addition, if *IPARM(9)* $=$ 2 on input, it is set to 1 on output. If *IPARM(9)* $= 1$ on input and ordering is one of the steps being performed, then *IPARM(9)* is treated as if it was equal to 2 on input. If a solution is obtained or iterative refinement is performed, then the solution is in the original unpermuted order if *IPARM(9)* $= 0$ and it is permuted if *IPARM(9)* is 1 or 2 on input. The input *RHS* is overwritten by the permuted *RHS* only if the triangular solutions are performed.

- **IPARM(10)^S, type M:**

  An input of *IPARM(10)* $= 0$ implies that *WSSMP* will not perform any scaling of the input matrix and vector(s).

  If *IPARM(10)* $= 1$, then a scaling of the form $(D^{-1}AD^{-1}Dx = D^{-1}b$ is performed, where $D = diag(\sqrt{a_{i,i}})$. The system $Hy = c$ is solved, where $H = D^{-1}AD^{-1}$ and $c = D^{-1}b$. The solution of the system, $\overline{y}$, is then scaled to compute the solution of the original system $\overline{x} = D^{-1}\overline{y}$. If *IPARM(10)* $= 1$, scaling is skipped is a very small or negative diagonal value is encountered (this is indicated by *IPARM(10)* $= 0$ on output).

  IPARM(10) $= 2$ instructs *WSSMP* to perform the scaling only if any diagonal element of the original matrix A lies outside the range *[DPARM(10)*, 1.0/*DPARM(10)]*. Upon return, *IPARM(10)* is set to 0 or 1 to indicate whether or not the scaling was performed. If *IPARM(10)* $= 2$ and the range of diagonal values permits scaling, then it proceeds even if a very small or negative diagonal value is encountered. Scaling is skipped only for the rows/columns with a small, zero, or negative diagonal. Thus, an input values of *IPARM(10)* $= 2$ and *DPARM(10)* $= 1.0$ permit scaling of semidefinite matrices.

  If scaling is performed, then the values of the input matrix in *AVALS* are changed if the input format is *not* MSR *and* if the matrix either already permuted or is permuted on output; i.e., if *IPARM(4)* is 0 or 2 and *IPARM(8)* is 1 or 2. If MSR input format is used or if *IPARM(8)* is 0, only an internal copy is scaled and the input matrix remains unchanged.

  This option is currently unavailable in the parallel version.

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

  IPARM(11) and *IPARM(12)* instruct *WSSMP* how to handle very small, zero, or negative pivots. If *IPARM(11)* $= 0$, then if *WSSMP* encounters a diagonal value less than or equal to *DPARM(11)* during factorization (just before the square root), it sets *IPARM(64)* to the index of this pivot, puts the actual pivot value in *DPARM(64)* and returns without further processing. If *IPARM(11)* = 1, then if a diagonal value less than *DPARM(11)* is encountered, then it is replaced by the square of the value in *DPARM(21)* (in other words, it is replaced by *DPARM(21)* and no square root is performed) and the factorization continues. An input value of *IPARM(11)* = 2 is treated similar to *IPARM(11)* = 1, except that in addition to replacing a bad pivot with *(DPARM(21))*^{2}, *MRP(I)* is set to the integer value in *IPARM(13)* to flag the occurrence of a bad I-th pivot. If *IPARM(11)* = 2 and *IPARM(4)* $\geq$ 8, then *IPARM(11)* takes precedence and *IPARM(4)* is set to *IPARM(4)* - 8. The number of occurrences of diagonal values less than or equal to *DPARM(11)* is reported in *IPARM(21)* on output.

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

  If *IPARM(12)* = 0, then *DPARM(12)* *DPARM(22)* are ignored. An input of *IPARM(12)* = 1 means that if the I-th pivot is greater than *DPARM(11)* but less than or equal to *DPARM(12)*, it will be replaced by *DPARM(22)*^{2} during factorization. The number of pivots encountered in the range > *DPARM(11)* and $\leq$ *DPARM(12)* is reported in *IPARM(22)* on output. If *DPARM(12)* < *DPARM(11)* and *IPARM(12)* = 1, then *DPARM(12)* is set to *DPARM(11)*.

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

  IPARM(13) is the integer flag used for marking bad pivots in the *MRP* array if *IPARM(11)* = 2.

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

  IPARM(14) = 0 is the default and has no effect.

  If *IPARM(14)* is 1, then *WSSMP* uses the space in *AVALS* to store a permuted internal copy of the matrix values. If the input format is MSR (i.e., *IPARM(4) = 1*), then this option should be used only if *AVALS* has the space for at least N extra double precision numbers. This *is not* recommended in parallel version as the original values are discarded anyway at a slightly later stage so copying then in *AVALS* wastes time.

  If *IPARM(14)* is 2, then *WSSMP* uses the space in *JA* to store a permuted internal copy of the matrix indices. If the input format is MSR (i.e., *IPARM(4) = 1*), then this option should be used only if *JA* has the space for at least N extra integers. This *is* recommended in parallel version as and saves space on processor 0 when the entire original matrix resides on processor 0.

  If *IPARM(14)* is 3, then both *AVALS* and *JA* are used to store a permuted internal copy of the matrix.

  If this option is being be used and iterative refinement is performed in a different call to *WSSMP* than the factorization call, then *AVALS* must be passed unaltered to the iterative refinement call.

  An alternate way to reuse the space in *AVALS* and *JA* when input format is 0 is to use *IPARM(8)* = 2. When input format is 0, this space is automatically reused if *IPARM(8)* = 1.

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

IPARM(15) contains the input parameter *N1*, where $0 \leq N1 \leq N$. *N1* is used only if *WSSMP* is used for ordering and is ignored if it is less than or equal to 0 or greater than or equal to N. If *N1* is used, then the first *N1* columns of the matrix are factored before the remaining $N-N1$ columns.

- **IPARM(16:20), type M:**

  The five consecutive integer values in *IPARM(16)* through *IPARM(20)* are identical to the five entries in the *OPTIONS* array in the calling sequence of the *WMMRB* routine for ordering. Please refer to Section 4.1.4 for details on these important options.

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

  Please refer to the description of *IPARM(11)*.

- **IPARM(22), type O:**

  Please refer to the description of *IPARM(12)*.

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

  On output, this contains the number of double words of storage required by Cholesky factorization to store the factor and as additional working space. This field is set only if symbolic factorization is performed.

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

  On output, *IPARM(24)* contains the number of nonzeros in the triangular factor. This field is set only if symbolic factorization is performed.

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

  This parameter indicates whether or not a condition number estimate is to be performed. If *IPARM(25)* $= 0$, then a condition number estimate is not computed. If *IPARM(25)* > 0, then the value of $||A||$ is computed and placed as output in *DPARM(2)* and an estimate of $||A^{-1}||$ is computed and placed in *DPARM(3)*. The condition number of the matrix A is the product *DPARM(2)* $\times$ *DPARM(3)*. The condition number is a measure of the reliability of the solution of a given linear system and can be used to compute error bounds. The algorithm described in [2] is used to estimate $||A^{-1}||$. If *IPARM(25)* $= 1$, then 1-norms are computed, if *IPARM(25)* $= 2$, then Frobenius-norms are computed, and if *IPARM(25)* $= 3$, then infinity-norms are computed. *WSSMP* can be made to do this computation either with factorization, or with solution, or with iterative refinement. *WSSMP* keeps track of the condition number estimation computation with respect to a particular norm for a given matrix and if *IPARM(25)* does not change between factorization, solution and iterative refinement steps, then the computation is not repeated.

  Note that if scaling is opted for (i.e., if *IPARM(10)* $= 1$) then the norms computed are those of the scaled matrix and not the original matrix.

  Condition number estimation is not yet available in the parallel version.

- **IPARM(26)^P, type I or M:**

This parameter is relevant only in the parallel version and specifies the block size that the internal dense matrix computations use for the two dimensional decomposition of the frontal and update matrices. If it is 0, then the parallel solver chooses an appropriate value and puts it in *IPARM(26)*, otherwise, it uses the largest power of 2 less than or equal to *IPARM(26)*.

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

  This parameter, which is relevant only in the parallel version, allows the user to control the load-balance versus fill-in trade-off to some extent. *IPARM(27)* = 0 is the default and allows the ordering to determine load-balancing among the processors with minimal intervention from *PWSSMP*. If *IPARM(27)* = 1, then *PWSSMP* performs somewhat aggressive restructuring of the elimination tree in order to improve load-balancing. In addition to the possibility of an increase in factorization fill-in, such restructuring also increases the time for symbolic factorization (because the restructuring is performed during symbolic factorization). Therefore, using this option is not recommended unless the graph of the sparse matrix is very irregular (which is when the restructuring is most beneficial) and the extra time spent in symbolic factorization can be recovered over many numerical factorizations of matrices with the same structure. The restructuring is usually beneficial for interior-point codes.

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

  This parameter is relevant only for the parallel version. If triangular solves with many right-hand sides are being performed (i.e., $NRHS \gg 1$), then *PWSSMP* partitions B along columns into blocks of 20 columns and solves for 20 right-hand sides at a time. If *IPARM(28)* = 0 on input, then the maximum default block size of 20 is used for partitioning B. If *IPARM(28)* > 0, then the value of *IPARM(28)* itself is used as the block size. If this block size is too large, then *PWSSMP* can run out of memory. On the other hand using a small block size for multiple right-hand sides increases the communication overhead of the triangular solves. This parameter is irrelevant for $NRHS = 1$.

- **IPARM(29:63), type R:**

  These are reserved for future use.

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

  In the event of a successful return from *WSSMP*, *IPARM(64)* is set to 0 on output. A nonzero value of *IPARM(64)* upon output indicates that *WSSMP* did not complete execution and detected an error condition. If the user supplies the memory for factorization in *AUX* and the size *NAUX* of this array is not sufficient, then *IPARM(64)* is -101 on output. If dynamic memory allocation by *WSSMP* fails (which can happen even if the user supplies factorization space in *AUX* because *WSSMP* allocates more memory for internal book-keeping), the *IPARM(64)* is set to -102 on return. An output value of -700 for *IPARM(64)* indicates an internal error and should be reported to *anshul@watson.ibm.com*. An output value of -200 for *IPARM(64)* in the parallel version indicates that the problem is too small for the given number of processors and must be attempted on fewer processors. A positive integer value of *IPARM(64)* between 1 and N on output indicates a computation error. In this case, *IPARM(64)* is the index of the first pivot that was less than or equal to *DPARM(11)* before computing the square root for Cholesky factorization. If C-style indexing is used and *IPARM(64)* > 0, then *IPARM(64)* is 1 + the index of the bad pivot. 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.

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

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

The entries *DPARM(26)* through *DPARM(63)* 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 as follows:

- **DPARM(2), type O:**

  Please refer to the description of *IPARM(25)*.

- **DPARM(3), type O:**

  Please refer to the description of *IPARM(25)*.

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

  On output, this contains the largest diagonal element encountered in the factorization process just before the square root. The maximum is taken before the diagonal value is tested for conditions described in the description of *IPARM(10,11)* and unless any diagonal entry $L(i,i)$ is replaced by a large value, *DPARM(4)* $= \max_{i=1,N} L(i,i)^2$.

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

  On output, this contains the smallest diagonal element encountered in the factorization process just before the square root. The maximum is taken before the diagonal value is tested for conditions described in the description of *IPARM(10,11)* and if the factorization proceeds normally without a diagonal entry $L(i,i)$ being replaced, *DPARM(5)* $= \min_{i=1,N} L(i,i)^2$.

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

  Iterative refinement is stopped if *IPARM(7)* > 0 and the relative error becomes less than *DPARM(6)*. *DPARM(6)* is not used if *IPARM(7)* $= 0$.

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

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

- **DPARM(10)^S, type I:**

  Please refer to the description of *IPARM(10)*.

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

  This is the lower threshold on value of a good diagonal value. If a diagonal value is less than or equal to *DPARM(11)*, then either an error is flagged or corrective action is taken as specified by *IPARM(11)*.

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

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

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

  Please refer to the description of *IPARM(11)*. It is the user's responsibility to initialize *DPARM(21)* with a valid positive double precision value.

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

  Please refer to the description of *IPARM(12)*. If *DPARM(21)* is being used; i.e., if *IPARM(12)* = 1, then it is the user's responsibility to initialize *DPARM(21)* with a valid positive double precision value.

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

  This contains the number of floating point operations required in Cholesky factorization.

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

  This contains the total number of floating point operations required in forward and backward elimination for all the right-hand sides. The operation count if for a single right-hand side.

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

  If *IPARM(11)* = 0, then *DPARM(64)* contains the value of the first pivot (if any) encountered that was less than or equal to *DPARM(11)*.

4 Subroutines Providing a Simpler Interface

In this section, we describe a simpler interface to *WSSMP*. This interface accepts the input in only CSR and MSR 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 *WSSMP* routine described in Section 3, *unless mentioned otherwise*. This interface is provided via the following subroutines:

4.1 WMMRB, WKKTORD, WN1ORD (ordering)

WMMRB (N, XADJ, ADJNCY, OPTIONS, NUMBERING, PERM, INVP, AUX, NAUX)

WKKTORD (N, XADJ, ADJNCY, OPTIONS, NUMBERING, PERM, INVP, AUX, NAUX)

WN1ORD (N, N1, XADJ, ADJNCY, OPTIONS, NUMBERING, PERM, INVP, AUX, NAUX)

WMMRB is the main ordering routine whose calling sequence is described below in detail. The *WKKTORD* routine has exactly the same calling sequence as *WMMRB* and should be used for ordering the systems in which the coefficient matrix K takes the following form:

$$K = \begin{pmatrix} D_1 & M^T \\ M & D_2 \end{pmatrix},$$

where M is an $n \times m$ sparse matrix, D_1 is an $m \times m$ diagonal matrix, D_2 is an $n \times n$ diagonal matrix, and $n + m = N$. The routine *WN1ORD* is also similar to *WMMRB*, except that *WN1ORD* has an extra parameter *N1*, where $0 \leq N1 \leq N$. Any permutation generated by *WN1ORD* is such that the first *N1* columns of the matrix are factored before the remaining $N-$ *N1* columns. *N1* is ignored if it is less than or equal to 0 or greater than or equal to N.

4.1.1 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 $Ax = b$.

4.1.2 XADJ, (type I): pointers into adjacency list

```
INTEGER XADJ (NUMBERING : N + NUMBERING)
```

XADJ(I) points to the starting location in *ADJNCY* of the indices of the vertices adjacent to vertex I in the undirected graph corresponding to the sparse matrix being ordered. The vertices adjacent to vertex $I + 1$ must be stored immediately after the vertices adjacent to vertex I in *ADJNCY*. Thus, *XADJ(I+1) - XADJ(I)* is the degree of vertex I in the graph.

4.1.3 ADJNCY, (type I): adjacency list

```
INTEGER ADJNCY(NUMBERING:XADJ(N+NUMBERING)-1)
```

ADJNCY contains the adjacency list of the graph. The vertices adjacent to vertex I+1 must be stored immediately after the vertices adjacent to vertex I.

Note that *XADJ* and *ADJNCY* are different from *IA* and *JA* in that a vertex is not considered adjacent to itself and therefore I is not included in *ADJNCY(XADJ(I):XADJ(i+1)-1)*. On the other hand, all diagonal entries must be non-zero; therefore, I must be included in *JA(IA(I):IA(I+1)-1)* in the CSR/CSC format (*IPARM(4)* = 0). Moreover, the adjacency list for a vertex I must contain all neighbors of I or all entries in row/column I of the matrix except I itself. In contrast, the index list stored in JA corresponding to row/column I contains only those indices of row/column I that are greater than or equal to I.

4.1.4 OPTIONS, (type I): ordering options

```
INTEGER OPTIONS(1:5)
```

OPTIONS is an integer array of size 5 that is used to pass various optional parameters to *WMMRB*. Note that when *WSSMP* or *PWSSMP* is used for ordering, the meaning of *IPARM(16:20)* is the same as that of *OPTIONS(1:5)*.

- **OPTIONS(1), type I:**

  If *OPTIONS(1)* = 0, then all default options are used and speed of 3 is chosen (see below for description of speed). If *OPTIONS(1)* = 1, 2, or 3, then the options described below are used instead of the defaults. In addition, the ordering speed and quality is determined by the integer value (speed) in *OPTIONS(1)*. Speed = 1 results in the slowest but best ordering, speed = 3 results in fastest but worst ordering, and speed = 2 makes *WMMRB* work at moderate speed and generate an ordering of intermediate quality.

- **OPTIONS(2), type I:**

  OPTIONS(2) 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 *OPTIONS(2)* greater than *N*. A value of 0 in this field lets *WMMRB* 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 this element of *OPTIONS* is 0.

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

  The default value of 0 in *OPTIONS(3)* has no effect. *OPTIONS(3)* = 1 forces *WMMRB* to 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 in Cholesky 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 very useful when one ordering is used for multiple factorizations. Note that this option forces *WMMRB* to produce 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. In case of insufficient memory, the *ADJNCY* may be overwritten (indicated by a -100 in *OPTIONS(4)* on output) or permuted. If a user finds *ADJNCY* to be altered and *OPTIONS(4)* unchanged on output, then *ADJNCY* has not been destroyed but the row (column) indices of the columns (rows) have simply been permuted and *ADJNCY* is still valid.

  Note that in the parallel routine *PWSSMP*, *IPARM(18)*, which is the equivalent of *OPTIONS(3)* is ignored and the minimum local fill ordering is not performed because it may hamper parallelism in factorization.

- **OPTIONS(4), type I or M:**

  On input, *OPTIONS(4)* contains a random number seed. One can use different values of the seed to force *WMMRB* 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.

On output, a -102 in *OPTIONS(4)* indicates that dynamic memory allocation failed, a -101 indicates that *NAUX* was not large enough (if it was nonzero), a -100 indicates that ordering completed, but *ADJNCY* was overwritten due to shortage of space, and any other negative integer -k indicates that an error was detected in the k-th input argument. If none of the above events occur, then *OPTIONS(4)* is unchanged.

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

  The input *OPTIONS(5)* lets the user communicate some known characteristics of the sparse matrix to *WMMRB* 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. Certain sparse matrices have a few rows/columns that are much denser than most of the rows/col- umns. Most sparse matrices used in interior-point problems in linear programming fall in this category. For such matrices, the integer value supplied in *OPTIONS(5)* should be 1 or 3. This instructs *WMMRB* to split the graph based on the high degree nodes before proceeding with the ordering. 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 or 3 should be used in *OPTIONS(5)*. This instructs *WMMRB* to construct a compressed graph before proceeding with the ordering.

  To summarize, *OPTIONS(5)* should contain 0 for default, 1 for a preprocessing based on high degree nodes, 2 for preprocessing the graph to compress it, and 3 if both of the above preprocessing steps are desired. Preprocessing the high degree nodes may result in an improvement in the quality of ordering (if the matrix is suitable for this), and preprocessing for compression may result in a much faster ordering if there are significant (enough to compensate for the time spent in performing compression) number of nodes with identical adjacencies.

Note 4.1 Recommended options for the serial version: *The recommended contents of* OPTIONS(1:5) *and* IPARM(16:20) *in the serial mode are (1,0,1,0,1) for interior-point algorithms, (1,0,1,0,2) for finite-element problems that can benefit from compression; i.e., there is reasonable fraction of vertices with multiple degrees of freedom, and (1,0,1,0,0) for other matrices. In the serial version, if only one factorization is performed for each ordering and a fast ordering is important, then* OPTIONS(2) *(or* IPARM(17)*) should contain $N+1$, where N is the dimension of the system. The ordering speed can be further increased by using a higher value (2 or 3) in* OPTIONS(1) *or* IPARM(16)*.*

Note 4.2 Recommended options for the parallel version: *The recommended contents of* IPARM(16:20) *in the parallel version are (1,0,0,0,1) for linear-programming problems, (1,0,0,0,2) for finite-element matrices that can benefit from graph compression, and (1,0,0,0,0) otherwise. The ordering process can be speedup up by making* IPARM(17) $= K$, *where* $K = N/p - 1$, N *is the dimension of the system and* p *is the number of processors being used. The ordering speed can be further increased by using a higher value (2 or 3) in* OPTIONS(1) *or* IPARM(16)*.*

4.1.5 NUMBERING, (type I): indexing options

```
INTEGER NUMBERING
```

This is same as *IPARM(11)* described in Section 3.2.14.

4.1.6 PERM, (type O): permutation vector

```
INTEGER PERM ( N )
```

The description is the same as in Section 3.2.6.

4.1.7 INVP, (type O): inverse permutation vector

```
INTEGER INVP ( N )
```

The description is the same as in Section 3.2.7.

4.1.8 AUX, (type O, I, or T): auxiliary storage

```
INTEGER AUX ( NAUX )
```

AUX is temporary integer workspace, which is used only if $NAUX > 0$. The user has the option of having the *WMMRB* routines allocate memory dynamically by specifying $NAUX = 0$, or by supplying an integer array *AUX* of size $NAUX > 0$ and having the *WMMRB* routines use *AUX* as the work space. The exact value of *NAUX* required is data dependents, but roughly, for *WMMRB* with not too irregular graphs, $30 \times N + 6 \times E + 30000$ should be sufficient. The memory requirement can be a lot less if compression is opted for and is achieved.

In general, using $NAUX = 0$ is somewhat more robust (especially for ordering LP matrices) because memory requirement is problem dependent and may exceed the above estimates. However, if the calling program has already allocated a large amount of memory which is free at the time of ordering, it may be worthwhile to pass that in to *WMMRB* for its temporary use because additional memory allocation by *WMMRB* may fail under such circumstances.

The quality and the run time of ordering are affected by the amount of memory available.

4.1.9 NAUX, (type I): size of user supplied auxiliary storage

```
INTEGER NAUX
```

If $NAUX = 0$, then *AUX* is NULL and *WMMRB* performs dynamic memory allocation. A positive integer value in *NAUX* forces *WMMRB* to use the first *NAUX* integer locations of *AUX* for its memory requirements.

4.2 WSCALZ (analyze, CSR input) and WSMALZ (analyze, MSR input)

WSCALZ (N, IA, JA, OPTIONS, PERM, INVP, NNZL, WSPACE, AUX, NAUX)

WSMALZ (N, IA, JA, OPTIONS, PERM, INVP, NNZL, WSPACE, AUX, NAUX)

The description of *OPTIONS*, *AUX*, and *NAUX* inputs is the same as in the calling sequence of *WMMRB* [5]. Note that *NAUX* for this routine is the number of *integers* that *AUX* can hold.

On output, the integer *NNZL* contains the number of nonzeros in the Cholesky factor and the integer *WSPACE* contains the number of double words of memory required by Cholesky factorization, including the storage for the Cholesky factor.

4.3 WSCSYM (symbolic, CSR input) and WSMSYM (symbolic, MSR input)

This routine performs symbolic factorization only and uses the input permutation given by *PERM* and *INVP*.

WSCSYM (N, IA, JA, PERM, INVP, NNZL, WSPACE)

WSMSYM (N, IA, JA, PERM, INVP, NNZL, WSPACE)

On output, the integer *NNZL* contains the number of nonzeros in the Cholesky factor and the integer *WSPACE* contains the number of *double words* of memory required by Cholesky factorization, including the storage for the Cholesky factor.

4.4 WSCCHF (Cholesky, CSR input) and WSMCHF (Cholesky, MSR input)

This routine performs Cholesky factorization, assumes that the input matrix is *not* permuted, and uses its permutation for factorization. The input *NAUX* should either be zero, or at least as large as the output *WSPACE* of the *analyze* or *symbolic routines*, either of which must be called before a *WSxCHF* routine.

WSCCHF (N, IA, JA, AVALS, PERM, INVP, AUX, NAUX, INFO)

WSMCHF (N, IA, JA, AVALS, DIAG, PERM, INVP, AUX, NAUX, INFO)

The description of the output *INFO* is the same is that of *IPARM(64)* in the calling sequence of the *WSSMP* routine described in Section 3.2.14.

4.5 WSCSVX and WSMSVX (expert drivers with CSR and MSR inputs)

These routines perform Cholesky factorization, triangular solutions, and iterative refinement. A call to the appropriate *WSxSYM* or *WSxALZ* routine must precede the call to a *WSxSVX* routine. *PERM* and *INVP* inputs determine the permutation of the input matrix that is used. The input matrix and *B* are assumed to be unpermuted and the output *B* is in the same order as the input.

WSCSVX (N, IA, JA, AVALS, PERM, INVP, B, LDB, NRHS, AUX, NAUX, RCOND, INFO)

WSMSVX (N, IA, JA, AVALS, DIAG, PERM, INVP, B, LDB, NRHS, AUX, NAUX, RCOND, INFO)

The double precision output *RCOND* is the inverse of the condition number estimate of the input matrix. The description of the output *INFO* is the same is that of *IPARM(64)* in the calling sequence of the *WSSMP* routine described in Section 3.2.14.

4.6 WSSLV (solve a system using a prior factorization)

The routine *WSSLV* solves a system of equations with right-hand side B, using a prior factorization by a call to either a *WSxCHF* or a *WSxSVX* routine.

WSSLV (N, PERM, INVP, B, LDB, NRHS, NITER, AUX, NAUX)

Note that if the user supplied *AUX* was used in factorization (i.e., $NAUX > 0$ in the call to the factorization routine) then the same *AUX* must be passed unaltered to *WSSMPSLV*. *NITER* is an input integer that specifies the number of iterative refinement steps to be performed. If *NITER* is 0, iterative refinement is not performed.

5 The Primary Parallel Subroutine: *PWSSMP*

The calling sequence for the parallel routine *PWSSMP* is identical to that of the serial routine *WSSMP* and the arguments have similar meanings. However, certain distinctions need to be made and the sizes of the arrays may need to be redefined. These distinctions are detailed in the following subsections.

5.1 Parallel modes and data-distribution

The parallel routine *PWSSMP* described in this section and the routines described later in Section 6 work in two modes; we shall call them the *0-master mode* and the *peer mode*. In the 0-master mode, all data including the entire matrix A and right-hand side b initially reside on processor 0; henceforth referred to as P_0. In general, if the program is running on p processors, we shall name the processors $P_0, P_1, \ldots, P_{p-1}$. In the peer mode, the initial data is distributed among the processors. In general, P_i initially owns N_i columns of the matrix A and also N_i positions of the right-hand side B. The dimension of the system of equations is $N = \Sigma_{i=0}^{p-1} N_i$. In the 0-master mode, $N = N_0$ and $N_1 = N_2 = \cdots = N_{p-1} = 0$. In the peer mode, $N > N_0$. There is no restriction on the relative amount of data on any of the processors in the peer mode and the entire matrix A residing on a single processor other than P_0 is a valid special case of the peer mode. Once again, the the parallel solver is in 0-master mode iff $N = N_0$ and in peer mode iff $N > N_0$.

Figure 2 illustrates the input data structures for the matrix A in the peer mode for $p = 3$, $N_0 = 3$, $N_1 = 2$, and $N_2 = 4$ for the matrix shown in Figure 1 earlier. Note that in the 0-master mode, processor 0 calls the parallel routine *PWSSMP* exactly the way it would call the serial routine and all the other processors call *PWSSMP* with $N_i = 0$. Therefore, the illustration of CSR and MSR (see Sections 3.1 and 3.2.2–5 for more details) in Figure 1 for the serial version is valid for the 0-master mode in the parallel version.

As shown in Figure 2, in the default peer mode, consecutive processors must contain consecutive portions of the matrix A (and also the right-hand side b). In other words, under default conditions, if l is the last column on processor P_i, then the first column on processor P_{i+1} must be $l + 1$. In addition, the indices and the values corresponding to consecutive columns must appear in consecutive order, just as in the serial version or the 0-master mode of the parallel version. However, in the peer mode, this default restriction can be overcome and any column of the the matrix can reside on any processor in any order if the special routine *WSETGLOBIND* is used. This routine is described in Section 7. Unless the *WSETGLOBIND* routine and the peer mode are used the global index of a

	P_0			P_1		P_2			
	1	2	3	4	5	6	7	8	9
1	14.		-1.				-1.	-3.	
2		14.	-1.					-3.	-1.
3	-1.	-1.	16.				-2.	-4.	-2.
4				14.		-1.	-1.	-3.	
5					14.	-1.		-3.	-1.
6				-1.	-1.	16.	-2.	-4.	-2.
7	-1.		-2.	-1.		-2.	16.	-4.	
8	-3.	-3.	-4.	-3.	-3.	-4.	-4.	71.	-4.
9		-1.	-2.		-1.	-2.		-4.	16.

A 9 X 9 symmetric sparse matrix.

A possible peer-mode distribution and the storage of this matrix in the input formats accepted by PWSSMP is shown in the table below:

	CSR Format				MSR Format				
Proc#	K	IA(K)	JA(K)	AVALS(K)	K	IA(K)	JA(K)	AVALS(K)	DIAG(K)
P_0	1	1	1	14.0	1	1	3	-1.0	14.0
	2	5	3	-1.0	2	4	7	-1.0	14.0
	3	9	7	-1.0	3	7	8	-3.0	16.0
	4	13	8	-3.0	4	10	3	-1.0	
	5		2	14.0	5		8	-3.0	
	6		3	-1.0	6		9	-1.0	
	7		8	-3.0	7		7	-2.0	
	8		9	-1.0	8		8	-4.0	
	9		3	16.0	9		9	-2.0	
	10		7	-2.0					
	11		8	-4.0					
	12		9	-2.0					
P_1	1	1	4	14.0	1	1	6	-1.0	14.0
	2	5	6	-1.0	2	4	7	-1.0	14.0
	3	9	7	-1.0	3	7	8	-3.0	
	4		8	-3.0	4		6	-1.0	
	5		5	14.0	5		8	-3.0	
	6		6	-1.0	6		9	-1.0	
	7		8	-3.0					
	8		9	-1.0					
P_2	1	1	6	16.0	1	1	7	-2.0	16.0
	2	5	7	-2.0	2	4	8	-4.0	16.0
	3	7	8	-4.0	3	5	9	-2.0	71.0
	4	9	9	-2.0	4	6	8	-4.0	16.0
	5	10	7	16.0	5	6	9	-4.0	
	6		8	-4.0					
	7		8	71.0					
	8		9	-4.0					
	9		9	16.0					

Figure 2: A sample distribution of the matrix in the two input formats for the peer mode of the parallel version.

column k on processor P_j is $k + \Sigma_{i=0}^{j-1} N_i$ (assuming that the columns are numbered from 1 through N).

5.2 Calling sequence

The parallel routine *PWSSMP* must be called on all the processors. The calling sequence on processor P_i is as follows ($0 \leq i < p$):

PWSSMP (N_i, IA_i, JA_i, $AVALS_i$, $DIAG_i$, PERM, INVP, B_i, LDB_i, NRHS, AUX_i, $NAUX_i$, MRP_i, IPARM, DPARM)

In the parallel version, an argument can be either *local* or *global*. A global array or variable must have the same size and contents on all processors and the size and contents of a local variable or array vary among the processors. In the context of *PWSSMP*, global does not mean globally shared, but refers to data that is replicated on all processors. In the above calling sequence, all arguments with a subscript are local.

Following is a brief description of the arguments. A more detailed description can be found in Section 3.2; this section is intended to highlight the differences between the serial and the parallel versions, wherever applicable.

- **$\mathbf{N}_i$:** The number of columns/rows of the matrix A and the number of rows of the right-hand side b residing on processor P_i. The total size N of system of equations is $\Sigma_{i=0}^{p-1} N_i$, where p is the number of processors being used.

  Note that, normally, the distribution chosen for a given matrix, cannot be changed between different phases. In other words, the N_i's must remain the same on each processor for each call made to *PWSSMP* in the context of the same system of equations. The only exception is when *IPARM(8)* $= 1$ in the peer mode for Cholesky factorization, then user is allowed to use a fresh distribution *only once* before the first Cholesky factorization for a given matrix. Please refer to the description of *IPARM(8)* below for more details.

- **$\mathbf{IA}_i$:** Integer array of size $N_i + 1$. This array provides pointers into the array of indices JA. See Figure 2 for more details. If $N_i = 0$, then this parameter can be a *NULL* pointer.

- **$\mathbf{JA}_i$:** Integer array of size $IA_i(N_i$ *+ IPARM(5)) - IPARM(5)* that contains the global row (column) indices of each column (row) on processor P_i. If $N_i = 0$, then this parameter can be a *NULL* pointer.

- **$\mathbf{AVALS}_i$:** Double precision array of size $IA_i(N_i$ *+ IPARM(5)) - IPARM(5)* that contains the numerical values corresponding to the indices in JA_i. If $N_i = 0$, then this parameter can be a *NULL* pointer.

- **$\mathbf{DIAG}_i$:** Double precision array of size N_i (only relevant if *IPARM(4)* $= 1$; i.e., MSR/MSC input format is used). If $N_i = 0$, then this parameter can be a *NULL* pointer.

- **PERM** *and* **INVP:** These two integer arrays contain the permutation and the inverse permutation vectors. The user of the parallel version must observe the differences with the serial version carefully for proper execution of their programs. Unlike most other parameters of *PWSSMP*, each processor must provide space (N integers) for full *PERM* and *INVP*. After the ordering step, only processor P_0 returns any meaningful information in *PERM* and *INVP*, although space for N integers must be provided in both *PERM* and *INVP* on all processors. The permutation in *PERM* and *INVP* after the ordering step is a valid fill-reducing ordering; however, it is altered during symbolic factorization. At the end os symbolic factorization, *PERM* and *INVP* and filled with a permutation on all processors and this is the permutation that is used in the remainder of the steps; i.e., Cholesky factorization, triangular solves, and iterative refinement. Just like in the serial routine *WSSMP*, a user of *PWSSMP* need not use the ordering produced by it, but can supply his/her own ordering. However, this user supplied ordering must be the input in *PERM* and *INVP* on processor P_0 for symbolic factorization and the modified ordering returned on all processors by symbolic factorization is the one that is actually used. Of course, the final ordering produced by the symbolic factorization phase is strongly related to the original ordering in *PERM* and *INVP* on P_0 before symbolic factorization, but it is not identical to it.

Note 5.1 *The parallel version is designed to automatically distribute the work among processors in accordance to their individual loads and speeds. This is because an IBM SP configuration can be heterogeneous with respect to the hardware on different nodes. As a side effect of this optimization, the vectors* PERM *and* INVP *may be different in different runs of the same problem on the same number of processors.*

- $\mathbf{B}_i$**,** $\mathbf{LDB}_i$**,** *and* **NRHS:** B is a double precision array of size $LDB_i \times NRHS$, where $LDB_i \geq N_i$. If $N_i = 0$, then B can be a $NULL$ pointer.
- $\mathbf{AUX}_i$ *and* $\mathbf{NAUX}_i$**:** If $NAUX_i > 0$, then AUX_i is a double precision array of size $NAUX_i$. If the user wishes to provide space in AUX_i for factorization, then the recommended minimum $NAUX_i$ is returned in *IPARM(23)* on each processor (a different value, in general) after symbolic factorization. Note that *PWSSMP* allocates storage dynamically in addition to that provided in AUX_i. If space is provided in AUX_i for factorization, then the same AUX_i must be passed unaltered to *PWSSMP* for triangular solves and iterative refinement.
- $\mathbf{MRP}_i$**:** Integer array of size N_i used only if *IPARM(11)* = 2. If $N_i = 0$, then *MRP* can be a $NULL$ pointer.
- **IPARM** *and* **DPARM:** All input parameters in these arrays must be identical on each processor. On output, *IPARM(21), IPARM(22), DPARM(4), DPARM(5), DPARM(7),* and *DPARM(64)* always identical on all processors. *IPARM(64)* is the same on all processors (returns number of the column on which factorization stopped) if it reports that factorization could not complete because a diagonal entry was too small or negative. If *IPARM(64)* reports another error, then the error, if any, is local to the reporting processors. The outputs in *IPARM(24), DPARM(23),* and *DPARM(24)* are reported only on P_0. The output *IPARM(23)* is local to each processor. *DPARM(2)* and *DPARM(3)*, which together report the condition number estimate in the serial version, always contain 0.0 on output in the parallel version because the parallel condition number estimator is not yet implemented.

  Similar to the serial version, it is legal for a user to have the input *IPARM(8)* set to 0 or 2 during ordering and symbolic factorization and to 1 in the remaining three computational phases. This allows the user to obtain the permutation (via vectors *PERM* and *INVP*) from

WSSMP or *PWSSMP* and generate a permuted matrix for factorization, solution, and iterative refinement. Often, this is more efficient than feeding in an unpermuted matrix, especially if the numerical values are being generated repeatedly (e.g., in an interior-point code) and the cost of generating a permuted or unpermuted matrix is more or less the same. In the parallel version, providing a permuted matrix can significantly improve factorization performance by improving the locality in the input data. If *IPARM(8)* is 1 for factorization in the peer mode, then the user is permitted to make the call to *PWSSMP* with different values of N_i than the ones used in previous *PWSSMP* calls for ordering and symbolic factorization. This is because permutation can change the distribution of the matrix. However, after the first *PWSSMP* call for factorization, the values of N_i should not change in any subsequent calls for factorization, triangular solves, or iterative refinement for the same system. This flexibility of redistributing the matrix does not allow the user to change from 0-master mode to peer mode, or vice-versa. If you are using the *WSETGLOBIND* routine (Section 7) to define a distribution of the matrix among the processors, then you permitted to make a call to *WSETGLOBIND* once before the first factorization to redefine the distribution in the peer mode while using *IPARM(8)* = 1.

5.3 Some parallel performance issues

The parallel version of the package performs ordering, factorization, solution, and iterative refinement in parallel. Symbolic factorization, which is the least time-consuming phase, is performed serially on processor P_0. Processor P_0 also bears a disproportionately high workload in the ordering phase compared to the other processors. SP installations are often heterogeneous with nodes varying in CPU's as well as the amount of memory on them. On a heterogeneous machine, the performance of the parallel routines will be much better if processor P_0 has a fast CPU, high memory bandwidth, and plentiful RAM. The time spent in ordering and symbolic factorization relative to that in the numerical computation phases increases as the number of processors increases because the numerical phases have a high scalability. The users can reduce the ordering and symbolic time at the cost of increased factorization time by carefully selecting the inputs in *IPARM(16), IPARM(17)*, and *IAPRM(27)*. See Sections 3.2.14 and 4.1.4 for details.

All parallel routines in the package utilize only a power-of-2 number of processors; i.e., 2, 4, 8, 16, 32, 64, and so on. Although the routines can be called on any number of processors with the initial data and the final results distributed on a subset or all of them, the performance on a given number of processors will be more or less the same as that on a number of processors that is the largest power of 2 less than or equal to the number of processors being used.

In most cases, using the peer mode with near uniform distribution of the input matrix (not in terms of N_i, but in terms of the sizes of arrays JA_i and $AVALS_i$) will yield much better performance than the 0-master mode. The latter however, is easier to use and provides a quick way of migrating from a serial to a parallel implementation because of identical calling sequences.

Providing a prepermuted matrix enhances parallel factorization performance and is recommended if the user can generate the permuted matrix cheaply. Please review the description of N_i and *IPARM(8)* in Section 5.2 carefully for the correct use of this option, especially in the peer mode. Please refer to the description of *IPARM(14)* for some more performance/memory issues relevant in the serial version and in the 0-master mode of the parallel version.

Since an IBM SP computer can have a heterogeneous configuration in which the nodes differ from each other in speed, the parallel version is designed to automatically detect heterogeneity and distribute the load accordingly, as far as possible. As a result, the performance of a p-processor heterogeneous configuration will usually be higher than that of a p-processor homogeneous configuration consisting of the slowest processors of the previous configuration. The performance of the solver on a heterogeneous configuration will vary depending on the relative location of fast and slow nodes in the system.

6 Parallel Subroutines Providing a Simpler Interface

In this section, we list the routines and that provide a simpler interface to *PWSSMP*. This interface is analogous to the simple interface to *WSSMP* described in Section 4. Parallel routines *PWSxALZ, PWSxCHF, PWSxSVX,* and *PWSSLV* are available to the users, where x is C for the CSR/CSC input format and M for the MSR/MSC input format. The function and the calling sequence of these routines are identical to the serial routines *WSxALZ, WSxCHF, WSxSVX,* and *WSSLV*, respectively described in Section 4. The meaning of the various parameters is the same as in the calling sequence of the *PWSSMP* routine described in Section 5.

7 Miscellaneous Routines

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

Both the serial and the parallel versions of the solver have the the context stored internally which enable them perform a desired task at any time while using the information from tasks performed earlier, provided that the necessary information was generated at least once. For examples, several calls to Cholesky factorization, triangular solves, and iterative refinement can be made with different data (but the same indices) 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 a routine *WSSFREE()* in the serial mode and *PWSSFREE()* in the parallel mode to free this storage if required. Both the routines have no arguments and can also be used with the simple interfaces described in Sections 4 and 6. After a call to any of these routines, the solver does not remember any context and the next call must be for performing ordering (or symbolic factorization if the user is providing his/her own *PERM* and *INVP*) to start a new context.

In the parallel version, by default it is assumed that the matrix is distributed on processors P_0 to P_{p-1} in increasing order of global indices and the portion of the matrix residing on processor P_i is determined by N_i on that processor (Figure 2). In the peer mode only of the parallel version, the user can specify an arbitrary distribution. This is accomplished by the use of the *WSETGLOBIND* routine described below. Note that the new global indices also determine the distribution of the right-hand side.

WSETGLOBIND (N_i, NUMBERING, GLI_i, INFO)

- The input parameter N_i is the number of matrix columns/rows on processor P_i. This N_i must be the same as the N_i in the subsequent calls to *PWSSMP* or any of the simpler routines of Section 6 in the context of the same matrix.

- The input parameter *NUMBERING* can be 0 or 1 depending on whether C-style indexing starting from 0 is used or FORTRAN-style indexing starting from 1 is used. This must be the same as *IPARM(5)* in subsequent calls to *PWSSMP* in the context of the same matrix. C-style numbering is not permitted with the simple routines of Section 6.

- GLI_i is an input array of size N_i. The k-th element of this integer array contains the global index of the k-th index on processor P_j. The default global index corresponding to the k-th index on P_j is $k + \Sigma_{i=0}^{j-1}$. The purpose of using the *WSETGLOBIND* subroutine is to override these defaults.

  For example, refer to the matrix shown in Figure 2. If the user wanted to use *PWSSMP* such that P_0 contains columns 6, 8, and 1 (in that order), P_1 contains columns 2, 3, and 7, and P_2 contains columns 9, 4, and 5, then *WSETGLOBIND* would be called on the three processors with $NUMBERING = 1$, $N_0 = 3$, $N_1 = 3$, $N_2 = 3$, $GLI_0 = (6,8,1)$, $GLI_1 = (2,3,7)$, and $GLI_2 = (9,4,5)$. In subsequent calls to the parallel routines, $JA_0 = (6,7,8,9,8,9,1,3,7,8)$, $JA_1 = (2,3,8,9,3,7,8,9,7,8)$, and $JA_2 = (9,4,6,7,8,5,6,8,9)$ in the CSR/CSC format. In this case, $IA_0 = (1,5,7,11)$, $IA_1 = (1,5,9,11)$, and $IA_2 = (1,2,6,10)$.

- The output parameter *INFO* is 0 on a successful return and is -102 if the routine runs out of memory.

8 Bugs and Problems

WSSMP, *PWSSMP*, as well as the routines in Sections 4 and 6 perform minimal input argument error-checking and it is the user's responsibility to call *WSSMP* subroutines with correct arguments, especially, valid options and symmetric 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, in addition to an invalid input argument, insufficient memory can also sometimes cause a routine to hang or crash before all the processor have a chance to return safely with an error report.

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

Version 2 of *WSSMP* is a beta release and is not guaranteed to be bug free. Any bugs or other problems with the software or this manual may be reported via e-mail to *anshul@watson.ibm.com.*

9 Notice: Terms and Conditions for Use of *WSSMP* and *PWSSMP*

IBM reserves the right to discontinue the free distribution of this software (the *WSSMP* and *PWSSMP* libraries) without any prior notice. IBM also reserves the right to reject a request for the software by any individual or organization. The information contained in this manual, including the calling sequences, 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 organizations and the U.S. government. 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 Corp.

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 program 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 it 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 *WSSMP* libraries is an implicit agreement by the user to these conditions. These terms and conditions are subject to change at any time without prior notice.

References

[1] E. Chu, Alan George, Joseph W.-H. Liu, and Esmond G.-Y. Ng. Users guide for SPARSPAK–A: Waterloo sparse linear equations package. Technical Report CS-84-36, University of Waterloo, Waterloo, IA, 1984.

[2] A. K. Cline, Cleve B. Moler, G. W. Stewart, and J. H. Wilkinson. An estimate for the condition number of a matrix. *SIAM Journal on Numerical Analysis*, 16:368–375, 1979.

[3] Anshul Gupta. *Analysis and Design of Scalable Parallel Algorithms for Scientific Computing.* PhD thesis, University of Minnesota, Minneapolis, MN, 1995.

[4] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix ordering. *IBM Journal of Research and Development*, 41(1/2):171–183, January/March, 1997.

[5] Anshul Gupta. WGPP: Watson graph partitioning (and sparse matrix ordering) package: Users manual. Technical Report RC 20453 (90427), IBM T. J. Watson Research Center, Yorktown Heights, NY, May 6, 1996.

[6] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5), May 1997.

[7] Mahesh Joshi, Anshul Gupta, George Karypis, and Vipin Kumar. Two dimensional scalable parallel algorithms for solution of triangular systems. Technical Report TR 97-024, Department of Computer Science, University of Minnesota, Minneapolis, MN, July 1997. A short version appears in *Proceedings of the 1997 International Conference on High Performance Computing (HiPC'97)*, Bangalore, India.

[8] Joseph W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. Technical Report CS-90-04, York University, Ontario, Canada, 1990. Also appears in *SIAM Review*, 34:82–109, 1992.