RC 21886 (98462) November 16, 2000 (Last update: January 2, 2021)
Computer Science/Mathematics

IBM Research Report

WSMP: Watson Sparse Matrix Package
Part I – direct solution of symmetric systems

Version 20.12

http://www.research.ibm.com/projects/wsmp

Anshul Gupta

IBM T. J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598

anshul@us.ibm.com

WSMP: Watson Sparse Matrix Package
Part I – direct solution of symmetric systems
Version 20.12

Anshul Gupta

IBM T. J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598

anshul@us.ibm.com

IBM Research Report RC 21886 (98462)

November 16, 2000

©IBM Corporation 1997, 2020. All Rights Reserved.

Contents

1 Introduction to Part I 5

2 Recent Changes and Other Important Notes 6

3 Obtaining, Linking, and Running *WSMP* 6
3.1 Libraries and other system requirements 6
3.2 License file 6
3.3 Linking on various systems 7
3.3.1 Linux on x86_64 platforms 7
3.3.2 Linux on Power 8
3.3.3 Cygwin on Windows 10 8
3.3.4 Mac OS 8
3.4 Controlling the number of threads 8
3.5 The number of MPI ranks per shared-memory unit 9

4 Overview of Functionality 9
4.1 Ordering 10
4.2 Symbolic factorization 10
4.3 Numerical factorization 10
4.4 Back substitution 11
4.5 Iterative refinement 11

5 The Primary Serial/Multithreaded Subroutine: *WSSMP* 11
5.1 Types of matrices accepted and their input format 11
5.2 Calling sequence of the WSSMP subroutine 12
5.2.1 N (type I): matrix dimension 13
5.2.2 IA (type I or M): row pointers 13
5.2.3 JA (type I or M): column indices 13
5.2.4 AVALS (type I or M): nonzero values of the coefficient matrix 14
5.2.5 DIAG (type I, O, or M): diagonal of coefficient or factor matrix 14
5.2.6 PERM (type I or O): permutation vector 14
5.2.7 INVP (type I or O): inverse permutation vector 14
5.2.8 B (type M): right-hand side vector/matrix 15
5.2.9 LDB (type I): leading dimension of B 15
5.2.10 NRHS (type I): number of right-hand sides 15
5.2.11 AUX (type O, I, or T): auxiliary storage 15
5.2.12 NAUX (type M): size of user supplied auxiliary storage 15
5.2.13 MRP (type O): pivot info 15
5.2.14 IPARM (type I, O, M, and R): integer array of parameters 16
5.2.15 DPARM (type I, O, M, and R): double precision parameter array 28

6 The Out-of-Core Solver: Using Secondary Storage 31
6.1 Environment Variables for OOC Computation 31
6.2 System Settings for OOC Computation 33
6.2.1 Linux 33
6.2.2 AIX 33
6.3 Possible Errors During OOC Computation 33

7 Subroutines Providing a Simpler Serial/Multithreaded Interface 33
7.1 WMMRB, WKKTORD, WN1ORD (ordering) . . . 34
7.1.1 N, (type I): matrix dimension . . . 34
7.1.2 XADJ, (type I): pointers into adjacency list . . . 34
7.1.3 ADJNCY, (type I): adjacency list . . . 34
7.1.4 OPTIONS, (type I): ordering options . . . 35
7.1.5 NUMBERING, (type I): indexing options . . . 36
7.1.6 PERM, (type O): permutation vector . . . 36
7.1.7 INVP, (type O): inverse permutation vector . . . 36
7.1.8 AUX, (type O, I, or T): auxiliary storage . . . 37
7.1.9 NAUX, (type I): size of user supplied auxiliary storage . . . 37
7.2 WSCSYM (symbolic, CSR input) and WSMSYM (symbolic, MSR input) . . . 37
7.3 WSCALZ (analyze, CSR input) and WSMALZ (analyze, MSR input) . . . 37
7.4 WSCCHF (Cholesky, CSR input) and WSMCHF (Cholesky, MSR input) . . . 38
7.5 WSCLDL and WSMLDL (LDL^T factorization, CSR and MSR inputs) . . . 38
7.6 WSCSVX and WSMSVX (expert drivers with CSR and MSR inputs) . . . 38
7.7 WSSLV (solve a system using a prior factorization) . . . 39

8 The Primary Message-Passing Parallel Subroutine: *PWSSMP* 39
8.1 Parallel modes and data distribution . . . 39
8.2 Calling sequence . . . 40
8.3 Some parallel performance issues . . . 42

9 Parallel Subroutines Providing a Simpler Interface 43

10 Miscellaneous Routines 43
10.1 *WS_SORTINDICES_I (M, N, IA, JA, INFO)* ^{S,T} . . . 43
10.2 *WS_SORTINDICES_D (M, N, IA, JA, AVALS, INFO)* ^{S,T} . . . 43
10.3 *WS_SORTINDICES_Z (M, N, IA, JA, AVALS, INFO)* ^{S,T} . . . 43
10.4 *WSETMAXTHRDS (NUMTHRDS)* . . . 44
10.5 *WSSYSTEMSCOPE* and *WSPROCESSSCOPE* . . . 44
10.6 *WSETMAXSTACK (FSTK)* . . . 44
10.7 *WSETLF (DLF)* ^{T,P} . . . 44
10.8 *WSETNOBIGMAL ()* . . . 45
10.9 *WSMP_VERSION (V, R, M)* . . . 45
10.10 *WSMP_INITIALIZE ()* ^{S,T} and *PWSMP_INITIALIZE ()* ^{P} . . . 45
10.11 *WSMP_CLEAR ()* ^{S,T} and *PWSMP_CLEAR ()* ^{P} . . . 45
10.12 *WSFFREE ()* ^{S,T} and *PWSFFREE ()* ^{P} . . . 46
10.13 *WSAFREE ()* ^{S,T} and *PWSAFREE ()* ^{P} . . . 46
10.14 *WSSFREE ()* ^{S,T} and *PWSSFREE ()* ^{P} . . . 46
10.15 *WSSMATVEC (N, IA, JA, AVALS, X, B, IERR)* ^{S} . . . 46
10.16 *PWSSMATVEC* (N_i, IA_i, JA_i, $AVALS_i$, X_i, B_i, *IERR*) ^{P} . . . 47
10.17 *WSTOREMAT (ID, INFO)* ^{S,T} and *PWSTOREMAT (ID, INFO)* ^{P} . . . 47
10.18 *WRECALLMAT (ID, INFO)* ^{S,T} and *PWRECALLMAT (ID, INFO)* ^{P} . . . 47
10.19 *WSETMPICOMM (INPCOMM)* ^{P} . . . 47
10.20 *WSADJSTBADPIVS (N, GAMMA)* ^{S,T} and *PWSADJSTBADPIVS (N, GAMMA)* ^{P} . . . 48
10.21 *WSETGLOBIND* (N_i, *NUMBERING*, GLI_i, *INFO*) ^{P} . . . 48
10.22 Routines for transposing distributed sparse matrices . . . 48
10.22.1 *PWS_XPOSE_IA* (N_i, IA_i, JA_i, NNZ_i, *IERR*) ^{P} . . . 49
10.22.2 *PWS_XPOSE_JA* (N_i, IA_i, JA_i, TIA_i, TJA_i, *IERR*) ^{P} . . . 49

10.22.3 *PWS_XPOSE_AV (*N_i*,* IA_i*,* JA_i*,* $AVALS_i$*,* TIA_i*,* TJA_i*,* $TAVALS_i$*, IERR)*^{P} 49
10.22.4 *PZS_XPOSE_AV (*N_i*,* IA_i*,* JA_i*,* $AVALS_i$*,* TIA_i*,* TJA_i*,* $TAVALS_i$*, IERR)*^{P} 49
10.22.5 *PWS_XPOSE_CLEAR ( )* 49

11 Routines for Double Complex Data Type **50**

12 Notice: Terms and Conditions for Use of *WSMP* and *PWSMP* **50**

13 Acknowledgements **50**

1 Introduction to Part I

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. It can be used as a in a shared-memory multiprocessor environment, or as a scalable parallel solver in a message-passing environment, where each MPI process can either be serial or multithreaded. *WSMP* is comprised of three parts. This document describes Part I for the direct solution of *symmetric* sparse systems of linear equations, either through LL^T factorization, or through LDL^T factorization. Part II uses sparse LU factorization with pivoting for numerical stability to solve general systems. Part III contains preconditioned iterative solvers. Parts II and III of User's Guide can be obtained from *http://www.research.ibm.com/projects/wsmp*, along with some example programs and technical papers related to the software. A current list of known bugs and issues is also maintained at this web site.

For solving symmetric positive definite systems, *WSMP* uses a modified version of the multifrontal algorithm [5, 16] for sparse Cholesky factorization and a highly scalable parallel sparse Cholesky factorization algorithm [11, 7, 12]. In a multithreaded environment, it assigns all threads to root of the elimination tree [15] (or a subtree that belongs to an MPI process) and recursively assigns the threads of each parent to its children in the tree to coarsely balance the load. The dense operations on the frontal matrices of tree vertices that are assigned more than one thread are also parallelized. The threads are managed through a task-parallel engine [14] that achieves fine-grain load balance via work-stealing. The package also uses scalable parallel sparse triangular solvers [13] and an improved and parallelized version of a multilevel nested-dissection algorithm [8] for computing fill-reducing orderings. Some details of the implementation and performance of *WSMP* for solving symmetric sparse systems are discussed by Gupta et al. [10]. For solving symmetric indefinite systems that require pivoting, WSMP uses 1×1 and 2×2 pivot blocks, as in the algorithm by Bunch and Kaufman [2]. The indefinite solver with pivoting is currently available only in serial and multithreaded modes.

The *WSMP* software is packaged into two libraries. The serial and multithreaded single-process routines are a part of the *WSMP* library. This library can be used on a single core or multiple cores on a shared-memory machine. The second library is called *PWSMP* and is meant to be used in the distributed-memory parallel mode. Each MPI process can itself be multithreaded.

The *WSMP* library can perform any or all of the following tasks: *ordering, symbolic factorization, Cholesky or* LDL^T *factorization, triangular solves,* and *iterative refinement*. The MPI-parallel version does not have out-of-core capabilities and the problems must fit in the main memory for reasonable performance.

The functionality and the calling sequences of the serial, multithreaded, and the message-passing parallel versions are almost identical. This document is organized accordingly and the descriptions of most parameters for both versions is included in the description of the combined serial and multithreaded version. The serial version supports certain features that the current message-passing 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 multithreaded version appear with a superscript ^{T} and those relevant to the message-passing parallel version appear with a superscript ^{P}.

Note 1.1 *Although* WSMP *and* PWSMP *libraries contain multithreaded code, the libraries themselves are* **not** *thread-safe. Therefore, the calling program cannot invoke multiple instances of the routines contained in* WSMP *and* PWSMP *libraries from different threads at the same time.*

The organization of this document is as follows. Section 2 describes important recent changes in the software that may affect the users of earlier versions. Section 3 lists the various libraries that are available and describe how to obtain and use the libraries. Section 4 gives an overview of the functionality of the serial and parallel routines for solving symmetric systems and explains how various functions performed by these routines affect each other. 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, and the differences from the message-passing parallel version are noted, wherever applicable. Section 6 describes the out-of-core shared-memory parallel solver. Section 7 describes user callable routines that provide a simpler interface to the serial and multithreaded solver, but omit some of the advanced features. Section 8 describes the two distributed modes in which the message-passing parallel solver can be used, describes the input data structures for the parallel solution,

and reminds users of the differences between the serial and the message-passing parallel versions, wherever applicable. This section does not repeat the information contained in Section 5 because the two user-interfaces are quite similar. Section 9 is the parallel analog of Section 7 and describes user callable routines that provide a simpler interface to the message-passing parallel solver. Section 10 describes a few utility routines available to the users. Section 11 gives a brief description of the double-complex data type interface of *WSMP*'s symmetric direct solvers. Section 12 contains the terms and conditions that all users of the package must adhere to.

2 Recent Changes and Other Important Notes

Versions 18 and later return the elapsed wall clock time for each call in *DPARM(1)* or *dparm[0]*.

Iterative solvers preconditioned with incomplete Choleski and LU factorization are now available. Please refer to the documentation for Part III, which can be found at *http://www.research.ibm.com/projects/wsmp*.

3 Obtaining, Linking, and Running *WSMP*

The software can be downloaded in gzipped tar files for various platforms from *www.research.ibm.com/projects/wsmp*.

If you need the software for a machine type or operating system other than those included in the standard distribution, please send an e-mail to *wsmp@us.ibm.com*.

The *WSMP* software is packaged into two libraries. The multithreaded library names start with *libwsmp* and the MPI_based distributed-memory parallel library names start with *libpwsmp*.

3.1 Libraries and other system requirements

The users are expected to link with the system's Pthread and Math libraries. In addition, the users are required to supply their own BLAS library, which can either be provided by the hardware vendor or can be a third-party code. The user must make sure that any BLAS code linked with *WSMP* runs in serial mode only. *WSMP* performs its own parallelization and expects all its BLAS calls to run on a single thread. BLAS calls running in parallel can cause substantial performance degradation. With some BLAS libraries, it may be necessary to set the environment variable *OMP_NUM_THREADS* to 1. Many BLAS libraries have their own environment variable, such as *MKL_NUM_THREADS* or *GOTO_NUM_THREADS*, which should be set to 1 if available.

On many systems, the user may need to increase the default limits on stack size and data size. Failure to do so may result in a hung program or a segmentation fault due to small stack size and a segmentation fault or an error code (*IPARM(64)*) of -102 due to small size of the data segment. Often the *limit* command can be used to increase *stacksize* and *datasize*. When the *limit* command is not available, please refer to the related documentation for your specific system. Some systems have separate hard and soft limits. Sometimes, changing the limits can be tricky and can require root privileges. You may download the program *memchk.c* from *www.research.ibm.com/projects/wsmp* and compile and run it as instructed at the top of the file to see how much stack and data space is available to you.

3.2 License file

The main directory of your platform contains a file *wsmp.lic*. This license file must be placed in the directory from which you are running a program linked with any of the *WSMP* libraries. You can make multiple copies of this file for your own personal use. Alternatively, you can place this file in a fixed location and set the environment variable WSMPLICPATH to the path of its location. *WSMP* first tries to use the *wsmp.lic* from the current directory. If this file is not found or is unusable, then it attempts to use *wsmp.lic* from the path specified by the WSMPLICPATH environment variable. It returns with error -900 in *IPARM(64)* if both attempts fail.

The software also needs a small scratch space on then disk and uses the */tmp* directory for that. You can override the default by setting the environment variable TMPDIR to another location.

3.3 Linking on various systems

The following sections show how to link with *WSMP* and *PWSMP* libraries on some of the platforms on which these libraries are commonly used. If you need the *WSMP* or *PWSMP* libraries for any other platform and can provide us an account on a machine with the target architecture and operating system, we may be able to compile the libraries for you. Please send e-mail to *wsmp@us.ibm.com* to discuss this possibility.

3.3.1 Linux on x86_64 platforms

Many combinations of compilers and MPI are supported for Linux on x86 platforms.

The most important consideration while using the distributed-memory parallel versions of *WSMP* on a Linux platform is that MPI library may not have the required level of thread support by default. The symmetric solver needs MPI_THREAD_FUNNELED support and the unsymmetric solver needs MPI_THREAD_MULTIPLE support. Therefore, MPI must be initialized accordingly. If MPI_THREAD_MULTIPLE support is not available, then you can use only one thread per MPI process. This can be accomplished by following the instructions in Section 10.4.

Note 3.1 *With most MPI implementations, when using more than one thread per process, the user will need to initialize MPI using* MPI_INIT_THREAD *(Fortran) or* MPI_Init_thread *(C) and request the appropriate level of thread support. The default level of thread support granted by using* MPI_INIT *or* MPI_Init *may not be sufficient, particularly for the unsymmetric solver. You may also need to use the* -mt_mpi *flag while linking with Intel MPI for the unsymmetric solver.*

Note 3.2 *There may be environment variables specific to each MPI implementation that need to be used for obtaining the best performance. Examples of these include* MV2_ENABLE_AFFINITY *with* mvapich2 *and* I_MPI_PIN, I_MPI_PIN_MODE, I_MPI_PIN_DOMAIN *etc. with Intel MPI.*

On all Linux platforms, under most circumstances, the environment variable MALLOC_TRIM_THRESHOLD_ must be set to -1 and the environment variable MALLOC_MMAP_MAX_ must be set to 0, especially when using the serial/multithreaded library. However, when using the message passing *PWSMP* library, setting MALLOC_TRIM_THRESHOLD_ to -1 can result in problems (including crashes) when more than one MPI process is spawned on the same physical machine or node. Similar problems may also be noticed when multiple instances of a program linked with the serial/multithreaded library are run concurrently on the same machine. In such situations, it is best to set MALLOC_TRIM_THRESHOLD_ to 134217728. If only one *WSMP* or *PWSMP* process is running on one machine/node, then MALLOC_TRIM_THRESHOLD_ = -1 will safely yield the best performance.

The *WSMP* libraries for Linux need to be linked with an external BLAS library. Some good choices for BLAS are MKL from Intel, ACML from AMD, GOTO BLAS, and ATLAS. Please read Section 3.1 carefully for using the BLAS library.

The x86_64 versions of the *WSMP* libraries are available that can be linked with Intel's Fortran compiler *ifort* or the GNU Fortran compiler *gfortran* (not *g77/g90/g95*). Note that for linking the MPI library, you will need to instruct *mpif90* to use the appropriate Fortran compiler. Due to many different compilers and MPI implementations available on Linux on x86_64 platforms, the number of possible combinations for the message-passing library can be quite large. If the combination that you need is not available in the standard distribution, please contact *wsmp@us.ibm.com*.

Examples of linking with *WSMP* using the Intel Fortran compiler (with MKL) and *gfortran* (with a generic BLAS) are as follows:

ifort -o <executable> <user source or object files> -Wl,–start-group $(MKL_HOME)/libmkl_intel_lp64.a $(MKL_HOME)/libmkl_sequential.a $(MKL_HOME)/libmkl_core.a -Wl,–end-group -lwsmp64 -L<path of *libwsmp64.a*> -lpthread

gfortran -o <executable> <user source or object files> <BLAS library> -lwsmp64 -L<path of *libwsmp64.a*> -lpthread -lm -m64

An example of linking your program with the message-passing library *libpwsmp64.a* on a cluster with x86_64 nodes is as follows:

mpif90 -o <executable> <user source or object files> <BLAS library> -lpwsmp64 -L<path of *libpwsmp64.a*> -lpthread

Please note that use of the sequential MKL library in the first example above. The x86_64 libraries can be used on AMD processors also. On AMD processors, ACML, GOTO, or ATLAS BLAS are recommended.

3.3.2 Linux on Power

Linking on Power systems is very similar to that on the x86_64 platform, except that a BLAS library other than MKL is required. The IBM ESSL (Engineering and Scientific Subroutine Library) is recommended for the best performance on Power systems.

3.3.3 Cygwin on Windows 10

The 64-bit libraries compiled and tested in the Cygwin environment running under Windows 7 and Windows 10 are available. An example of linking in Cygwin is as follows (very similar to what one would do on Linux):

gfortran -o <executable> <user source or object files> -L<path of *libwsmp64.a*> -lwsmp -lblas -lpthread -lm -m64

Please refer to Section 3.4 to ensure that BLAS functions do not use more than one thread on each MPI process.

3.3.4 Mac OS

MAC OS libraries are available for Intel and GNU compilers. The BLAS can be provided by either explicitly linking MKL (preferred) or by using the *Accelerate* framework. Linking examples are as follows:

gfortran -o <executable> <user source or object files> -m32 -lwsmp -L<path of *libwsmp.a*> -lm -lpthread -framework Accelerate

gfortran -o <executable> <user source or object files> -m64 -lwsmp64 -L<path of *libwsmp64.a*> -lm -lpthread -framework Accelerate

Once again, it is important to ensure that the BLAS library works in the single-thread mode when linked with *WSMP*. This can be done by using the environment variables OMP_NUM_THREADS, MKL_NUM_THREADS, or MKL_SERIAL.

3.4 Controlling the number of threads

WSMP (or a *PWSMP* process) automatically spawns threads to utilize all the available cores that the process has access to. The total number of threads used by *WSMP* is usually the same as the number of cores detected by WSMP. The unsymmetric solver may occasionally spawn a few extra threads for short durations of time. In many situations, it may be desirable for the user to control the number of threads that *WSMP* spawns. For example, if you are running four MPI processes on the same node that has 16 cores, you may want each process to use only four cores in order to minimize the overheads and still keep all cores on the node busy. If *WSMP_NUM_THREADS* or *WSMP_RANKS_PER_NODE* (Section 3.5) environment variables are not set and *WSETMAXTHRDS* function is not used, then, by default, each MPI process will use 16 threads leading to thrashing and loss of performance.

Controlling the number of threads can also be useful when working on large shared global address space machines, on which you may want to use only a fraction of the cores. In some cases, you may not want to rely on WSMP's automatic determination of the number of CPUs; for example, some systems with hyper-threading may report the number of hardware threads rather than the number of physical cores to WSMP. This may result in an excessive number of threads when it may not be optimal to use all the hardware threads.

WSMP provides two ways of controlling the number of threads that it uses. You can either use the function *WSETMAXTHRDS (NUMTHRDS)* described in Section 10.4 inside your program, or you can set the environment variable *WSMP_NUM_THREADS* to *NUMTHRDS*. If both *WSETMAXTHRDS* and the environment variable

WSMP_NUM_THREADS are used, then the environment variable overrides the value set by the routine *WSETMAX-THRDS*.

3.5 The number of MPI ranks per shared-memory unit

While it is beneficial to use fewer MPI processes than the number of cores on shared-memory nodes, it may not be optimal to use only a single MPI process on highly parallel shared-memory nodes. Typically, the best performance is observed with 2–8 threads per MPI processes. When multiple MPI ranks belong to each physical node, specifying the number of ranks per node by setting the environment variable *WSMP_RANKS_PER_NODE* would enable WSMP to make optimal decisions regarding memory allocation and load-balancing. If the number of threads per process is not explicitly specified, then *WSMP_RANKS_PER_NODE* also lets *WSMP* figure out the appropriate number of threads to use in each MPI process.

In addition, the way the MPI ranks are distributed among physical nodes can have a dramatic impact on performance. The ranks must always be distributed in a block fashion, and not cyclically. For example, when using 8 ranks on four nodes, ranks 0 and 1 must be assigned to the same node. Similarly, ranks 2 and 3, 4 and 5, and 6 and 7 must be paired together.

Note that the *WSMP_RANKS_PER_NODE* environment variable does not affect the allocation of MPI processes to nodes; it merely informs *PWSMP* how the ranks are distributed. *PWSMP* does not check if the value of *WSMP_RANKS_PER_NODE* is correct.

4 Overview of Functionality

WSSMP and *PWSSMP* are the primary routines for solving symmetric sparse systems of linear equations. Both the serial/multithreaded and the message-passing parallel libraries allow the users to perform any appropriate subset of the following tasks: (1) Ordering, (2) Symbolic factorization, (3) Numerical factorization, (4) Back substitution, and (5) Iterative refinement. These functions can either be performed by calls to the primary serial and parallel subroutines *WSSMP* and *PWSSMP* (described in Sections 5 and 8, respectively), or by using the simpler serial and parallel interfaces (described in Sections 7 and 9, respectively). When using *WSSMP* or *PWSSMP* routines, *IPARM(2)* and *IPARM(3)* control the subset of the tasks to be performed (see Section 5 for more details). When using the simple interfaces, the tasks or the subsets of tasks to be performed are determined by the name of the routine. Note that users can supply their own ordering and skip *WSSMP*'s or *PWSSMP*'s ordering phase. Please see Section 4.1 for more details.

An any time during the process of solution of a sparse symmetric system of linear equations, the corresponding context is stored internally. For example, when a call to *WSSMP* or *PWSSMP* for back substitution is made, the software uses information generated and internally stored during the ordering, symbolic, and numerical factorization of the coefficient matrix to correctly solve the system of equations. A call to ordering or symbolic factorization with a valid input in *PERM* and *INVP* automatically signals the beginning of work on a new system and discards the old context. Therefore, by default, *WSSMP* and *PWSSMP* work on one system of equations at a time. However, there is a provision of storing and recalling a given context, thereby providing a mechanism for *WSSMP* and *PWSSMP* to work on multiple systems of equations together. Up to 64 different contexts can be stored, which can correspond to 64 different sparse linear systems in possibly different stages of solution. This mechanism is provided by *WSTOREMAT*, *PWSTOREMAT*, *WRECALLMAT*, and *PWRECALLMAT* routines, which are described in detail in Section 10. At a given time, only one context is active, and unless specifically mentioned, the functions of all the routines described in this document are confined to the sparse matrix/system whose context is currently active.

The *WSSMP* and *PWSSMP* routines perform minimal input argument error-checking and it is the user's responsibility to call *WSMP* subroutines with correct arguments and 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 message-passing parallel version, on rare occasions, insufficient memory can also cause a routine to hang or crash before all the processes/threads have had a chance to return safely with an error report. However, unlike the input argument and memory related errors, the numerical error checking capabilities of the computational routines are quite robust.

All *WSMP* routines can be called from Fortran as well as C or C++ programs using a single interface described in this document. As a matter of convention, symbols (function and variable names) are in capital letters in context of Fortran and in small letters in context of C. Please refer to Notes 5.3, 5.4, and 10.1 for more details on using *WSSMP* with Fortran or C programs.

In the following subsections, we describe the key functions and the interdependencies of the five tasks mentioned above. These functions and dependencies are valid for both interfaces—the *WSSMP* interface and the simple interface.

4.1 Ordering

The ordering routines take the indices of the matrix and some control integers as input and generate a symmetric permutation of the input matrix. This permutation is designed to minimize fill during factorization and to provide ample parallelism and load-balance during message-passing or multithreaded parallel factorization. For LDL^T factorization with diagonal pivoting, the ordering routines, in addition to the structure, examine the values of the coefficients as well in order to determine a symmetric permutation that balances fill reduction with numerical stability. Multiple factorizations with the same ordering can still be performed with different values. The original matrix is not altered; the permutation is stored in the vectors *PERM* and *INVP*. Please refer to Sections 5.2.6 and 5.2.7 for a detailed description of these vectors.

If pivoting is not to be performed during factorization, then the user may use a permutation from another source and need not use the *WSMP* libraries to generate the permutation. However, valid permutation vectors *PERM* and *INVP* must be passed on to symbolic factorization. If *PERM* and *INVP* vectors contain a valid ordering/permutation, then the *WSMP*'s ordering step can be skipped and the computation can start with the symbolic factorization step.

In the message-passing case, valid entries in *PERM* and *INVP* vectors are produced by the ordering phase and consumed by the symbolic factorization phase on process 0 only. If an external ordering/permutation is being supplied, then *PERM* and *INVP* vectors on process 0 must contain the entire valid permutation before the call for symbolic factorization.

4.2 Symbolic factorization

Symbolic factorization sets up the internal data structures to be used by the subsequent numerical factorization and solve phases. Most of the work performed by symbolic is invisible to the user, except some output information on the memory and computational requirements of the numerical phases to follow.

A notable side-effect of the symbolic phase is that it makes alterations to *PERM* and *INVP* and in the message-passing case, distributes the altered *PERM* and *INVP* to all the participating processes. In other words, *PERM* and *INVP* (generated either by *(P)WSSMP*'s ordering phase, or from an external source) are input for symbolic factorization and are modified. The *PERM* and *INVP* vectors that are produced as the output of symbolic factorization must be passed unaltered to all subsequent numerical factorization, solve, and iterative refinement steps.

4.3 Numerical factorization

The numerical factorization performs either LL^T or LDL^T factorization on the input matrix. LDL^T factorization can be performed with or without diagonal pivoting. A symbolic factorization step must have been performed for a matrix with identical nonzero pattern before the first call to numerical factorization. Once a symbolic factorization step has been performed, numerical factorization can be called any number of times for matrices with identical nonzero pattern but possibly different numerical values. The input matrix that is stored in *IA*, *JA*, and *AVALS* and passed to numerical factorization can either be the original matrix as is, or the permuted matrix generated by applying the *PERM* and *INVP* output of symbolic factorization to the original matrix. This is controlled by *IPARM(8)*, as described in Section 5.2.14.

Once a symbolic factorization 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*.

4.4 Back substitution

The back substitution or the triangular solve phase generates the actual solution to the system of linear equations. This phase requires the factors generated by a previous call to numerical factorization. After the coefficient matrix has been factored, 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.

4.5 Iterative refinement

Iterative refinement can be used to improve the solution produced by the back-substitution phase. The option of using extended precision arithmetic for iterative refinement is available. For many problems, this step is not necessary.

5 The Primary Serial/Multithreaded 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*, LL^T *or* LDL^T *factorization*, *forward and backward solve*, and *iterative refinement*. 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 5.2). 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 5.2.14 for more details.

In addition to the advanced interface that the *WSMP* 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 7.

5.1 Types of matrices accepted and their input format

The *WSSMP* routine works for symmetric positive-definite, quasi-definite, and indefinite matrices, with or without diagonal pivoting. For certain kinds of indefinite systems, special ordering techniques can be used (see Section 7.1 for more details) to avoid pivoting. *WSSMP* can also handle semi-definite matrices without pivoting via appropriate use of the options provided by *IPARM(11:13)*, and *DPARM(11,12,21,22)*.

If diagonal pivoting is necessary, then the user can choose pivoting threshold *DPARM(11)* and set *IPARM(31)* to trigger the use of the Bunch-Kaufman algorithm [2]. A full implementation of this option is not available in the message-passing version, although a highly effective partial implementation is available.

Note 5.1 *It is extremely important that all rows/columns of the coefficient matrix have a diagonal entry. If the diagonal entry in a certain row/column is zero (as the case may be in an indefinite or semi-definite system), there should be an explicit zero at the diagonal location.*

All floating point values must be 8-byte real numbers. All integers must be 4 bytes long unless you are using *libwsmp8_8.a*, which takes 8-byte integer inputs.

Since the input matrix is symmetric, only a triangular part is accepted as 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-UT) or the lower triangular part in the compressed sparse column (CSC-LT) format. The second format is modified compressed sparse row/column (MSR or MSC). The performance of *WSSMP* and *PWSSMP* is slightly better with the CSR/CSC format than with the MSR/MSC format. Figure 1 illustrates both input formats; they are also explained briefly in Sections 5.2.2, 5.2.3, 5.2.4, and 5.2.5.

Note 5.2 *The symmetric solver handles complex Hermitian matrices, although, technically they are not symmetric. In order to process Hermitian matrices correctly, they must be input in CSC-LT or MSC-LT formats only.*

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

	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.

Figure 1: Illustration of the two input formats for the serial/multithreaded *WSSMP* routines.

WSSMP 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 5.2.14 for more details.

5.2 Calling sequence of the WSSMP subroutine

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 *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 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 a sparse symmetric 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* overwrites B when *WSSMP* is called to compute the solution of the system.

Note 5.3 *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.

Note 5.4 *All user-callable* WSMP *and* PWSMP *routines expect their parameters to be passed by reference. Therefore, when calling* WSSMP *from a C program, the addresses of the parameters described in Section 5.2 must be passed.*

The calling sequence and descriptions of the parameters of *WSSMP* are as follows. Note that all arguments are not accessed in all phases of the solution process. The descriptions that follow indicate when a particular argument is not accessed. When an argument is not accessed, a NULL pointer or any scalar can be passed as a place holder for that argument. The example program *wssmp_ex1.f* at the *WSMP* home page illustrates the use of the *WSSMP* subroutine for the matrix shown in Figure 1.

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

void *wssmp_(int *n, int ia[], int ja[], double avals[], double diag[], int perm[], int invp[], double b[], double *ldb, int *nrhs, double *aux, int *naux, int mrp[], int iparm[], double dparm[])*

5.2.1 N (type I): matrix dimension

```
INTEGER N
int *n
```

This is the number of rows and columns in the sparse matrix A or the number of equations in the sparse linear system $AX = B$. It must be a nonnegative integer.

5.2.2 IA (type I or M): row pointers

```
INTEGER IA (N + 1)
int ia[]
```

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*. Note that empty columns (or rows) are not permitted; i.e., IA($i+1$) must be greater than IA(i).

Please refer to Figure 1 and description of *IPARM(4)* in Section 5.2.14 for more details. Section 8.2 contains more details for the requirements on *IA* in the distributed-memory parallel case.

5.2.3 JA (type I or M): column indices

```
INTEGER JA ( * )
int ja[]
```

The integer array *JA* contains 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, these consecutive column (row) indices of a row (column) *must* be sorted in increasing order upon input. *WSMP* provides two utility routines to sort the indices (see Section 10 for details). If CSR/CSC 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.

Please refer to Note 5.1 in Section 5.1. As a result, in CSR and CSC formats, N is a lower bound on the size of JA.

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

```
DOUBLE PRECISION AVALS ( * )
double 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. Also, please refer to Note 5.1 in Section 5.1. As a result, in CSR and CSC formats, N is a lower bound on the size of AVALS.

The input *AVALS* may be modified if scaling is performed (by setting *IPARM(10)* appropriately) *and* if *IPARM(8)* is not 0. 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 is either already permuted or is permuted on output; i.e., if *IPARM(4)* is 0 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.

5.2.5 DIAG (type I, O, or M): diagonal of coefficient or factor matrix

```
DOUBLE PRECISION DIAG ( N )
double diag[]
```

If the MSR input format is used, then *DIAG(I)* contains the I-th diagonal element of the coefficient matrix. If CSR/CSC format is 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).

If *IPARM(32)* is 1, then *DIAG* contains the diagonal of the factor upon output. Please refer to the description of *IPARM(32)* for more details, including Note 5.15.

DIAG is accessed only in the factorization phase if either *IPARM(4)* is 1 or if *IPARM(32)* is 1 and the factorization is being performed without pivoting. For LDL^T factorization with pivoting, $DIAG$ is a read-only array when the input format is MSR.

5.2.6 PERM (type I or O): permutation vector

```
INTEGER PERM ( N )
int perm[]
```

PERM is the permutation vector, as defined in Sparspak [3]. 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 5.2.14 for more details.

Note 5.5 *If ordering and symbolic factorization are performed in different calls, or an external non-WSMP ordering is used, then, in general, the contents of* PERM *and* INVP *are altered during symbolic factorization. This permutation produced at the end of the symbolic phase is the actual permutation that is used in the factor and solve stages. It is different (but similar in properties) to the permutation on* P_0 *at the beginning of symbolic factorization.*

5.2.7 INVP (type I or O): inverse permutation vector

```
INTEGER INVP ( N )
int invp[]
```

INVP is the inverse permutation vector, as defined in Sparspak [3]. 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 5.2.14 for more details. Also refer to Note 5.5.

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

```
DOUBLE PRECISION B ( LDB, NRHS )
double b[]
```

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. During the solution, X overwrites B. If the solve (Task 4) and iterative refinement (Task 5) are performed separately, then the output of the solve phase is the input for iterative refinement. B is accessed only in the triangular solution and iterative refinement phases.

5.2.9 LDB (type I): leading dimension of B

```
INTEGER LDB
int *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. Even if *NRHS* $= 1$, *LDB* must be greater than 0.

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

```
INTEGER NRHS
int *nrhs
```

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

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

```
DOUBLE PRECISION AUX ( NAUX )
double *aux
```

This argument is obsolete and will be deleted in the near future. Currently, its only purpose is backward compatibility. A dummy argument or a NULL pointer may be passed.

5.2.12 NAUX (type M): size of user supplied auxiliary storage

```
INTEGER NAUX
int *naux
```

This argument is obsolete and will be deleted in the near future. Currently, its only purpose is backward compatibility. A dummy argument or a NULL pointer may be passed.

5.2.13 MRP (type O): pivot info

```
INTEGER MRP ( N )
int mrp[]
```

MRP is accessed for real matrices only, if *IPARM(11)* = 2. *MRP* is not used for complex matrices.

For factorization without pivoting, if *IPARM(11)* is 2 on input, then on return from factorization, *MRP(I)* is set to *IPARM(13)* if the I-th pivot was less than or equal to *DPARM(10)* during factorization. Otherwise, *MRP(I)* is not touched. Please refer to the description of *IPARM(11)* for more details.

For LDL^T factorization with pivoting, the role of *MRP* is similar. In this case, all entries of *MRP* are overwritten during ordering. The entries corresponding to rows and columns all whose entries were less than or equal to *DPARM(10)*

during factorization contain a negative integer on output. The remaining entries contain nonnegative integers. Thus, if the rank M of the $N \times N$ matrix is less than N, then $M - N$ entries in *MRP* corresponding to a set of rows and columns linearly dependent on a subset of the remaining M rows and columns is marked by negative integers, while the remaining entries in *MRP* contain non-negative integers. Note that *IPARM(21)* returns $M - N$ if the user sets *IPARM(11)* to 1 or 2, where M is the rank of the matrix. Structural singularity is usually detected during ordering and numerical singularity during numerical factorization. After ordering, *IPARM(21)* returns $M - N$ corresponding to the structural rank M, which may be further reduced during numerical factorization.

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

```
INTEGER IPARM ( 64 )
int 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 a call to *WSSMP*. If *IPARM(1)* is 0, then *WSSMP* 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 *WSSMP* 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 *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 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.

Note that all reserved entries; i.e., *IPARM(36:63)* must be filled with 0's.

- **IPARM(1) or iparm[0], 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) or iparm[1], 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. Also, if *WSSMP* is called to perform multiple tasks in the same call and it returns with an error code in *IPARM(64)*, then the output in *IPARM(2)* indicates the task that failed. 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 or LDL^T Factorization
Task 4:	Forward and Backward Elimination
Task 5:	Iterative Refinement

- **IPARM(3) or iparm[2], 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.

Index	IPARM			DPARM		
	Default	Description	Type	Default	Description	Type
1	mandatory I/P	default/user defined	M	-	Elapsed time	O
2	mandatory I/P	starting task	M	-	matrix norm^{S,T}	O
3	mandatory I/P	last task	I	-	mat. inv. norm^{S,T}	O
4	0	I/P format	I	-	max. fact. diag.	O
5	1	numbering style	I	-	min. fact. diag.	O
6	1	max # of iter. ref.	M	2×10^{-15}	rel. err. lim.	I
7	3	residual norm type	I	-	rel. resid. norm	O
8	0	mat. permut. option	I	-	unused	-
9	0	RHS permut. option	I	-	unused	-
10	0	scaling option	I	10^{-18}	singularity threshold	I
11	0	bad pivot handling	I	0.001	pivot threshold	I
12	0	small piv. handling	I	4×10^{-16}	small piv. thresh.	M
13	0	bad pivot flag/# perturbs.	I/O	-	no. of supernodes	O
14	0	AVALS reuse opt.	M	-	unused	-
15	0	ordering restriction	I	0.0	reordering flag	M
16	1	ordering option 1	I	-	unused	-
17	0	ordering option 2	I	-	unused	-
18	1	ordering option 3	I	-	unused	-
19	0	ordering option 4	I	-	unused	-
20	0	ordering option 5	I	-	unused	-
21	-	bad pivot count	O	10^{200}	bad pivot subst.	I
22	-	-ve eigenvalue count	O	2×10^{-8}	small piv. subst.	I
23	-	factor memory	O	-	actual factor ops.	O
24	-	factor nonzeros	O	-	predicted factor ops.	O
$25^{S,T}$	0	cond. no. option	I	-	unused	-
26^{P}	0	block size	M	-	unused	-
$27^{P,T}$	0	balancing option	I	-	unused	-
28^{P}	0	tri. solve blk.	I	-	unused	-
29	0	garbage collection	I	-	unused	-
30	0	tri. solve opt.	I	-	unused	-
31	0	LL^T or LDL^T factor	I	0.0	Complex matrix with real non-diag.	I
32	0	diag. O/P opt.	I	-	unused	-
33	-	no. of CPUs used	O	-	unused	-
34	0	partial solution	I	-	unused	-
35^{P}	0	enable dynamic dist.	I	-	unused	-
$36^{S,T}$	0	enable OOC computation	M	-	unused	-
37-63	0	reserved	R	0.0	reserved	R
64	-	return err. code	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*. Please refer to the text for details on ordering options *IPARM(16:20)*. *IPARM(36:63)* must be filled with all 0's and *DPARM(36:63)* must all contain 0.0 on input.

- **IPARM(4) or iparm[3], type I:**

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

- **IPARM(5) or iparm[4], 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) or iparm[5], 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. On output, *IPARM(6)* contains the actual number of iterative refinement steps 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 has been achieved.

  The default value of *IPARM(6)* is 1 for the symmetric solver.

- **IPARM(7) or iparm[6], 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 relative residual norm. 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 residual norm 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 residual norm, 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 residual norm given by $\|b - A\overline{x}\|/\|b\|$ at the end of the last iterative refinement step is placed in *DPARM(7)*. In *WSSMP*, even if iterative refinement is not performed (i.e. if *IPARM(6)* = 0 or if *IPARM(3)* < 5) the value of relative residual norm is computed and placed in *DPARM(7)* if *IPARM(7)* = 1, 2, 3, 5, 6, or 7. However, in *PWSSMP*, relative residual norm is computed only if the iterative refinement step is performed. Relative residual norm with *PWSSMP* can be computed without actually performing iterative refinement by setting *IPARM(6)* (the number of iterative refinement steps) to 0.

  If *NRHS* > 1, then the maximum of the relative residual norms amongst the *NRHS* solution vectors is considered. Also note that, if scaling is performed (by setting *IPARM(10)* appropriately), then the relative residual norms are computed with respect to the scaled system and not the original system.

  The default value of *IPARM(7)* is 3.

Note 5.6 *Please note that the residual is computed at the end of the solution step (Step 4). Even though* IPARM(7) *pertains to iterative refinement, it must be set to the appropriate value before the triangular solution step to be effective.*

Note 5.7 *Computing the residual adds a small overhead to the solution. Therefore, when solving a large number of linear systems w.r.t. the same factor without iterative refinement,* IPARM(7) *should be set to 0 to switch the residual computation off. This is important in applications in which the triangular solve time dominates.*

Note 5.8 *In the message-passing parallel version, the relative residual norm is not computed and iterative refinement is not performed if* NRHS > 20. *So if more than 20 solutions are required with iterative refinement or relative residual norm computation, then these must be performed in batches of at most 20 each.*

- **IPARM(8) or iparm[7], type I:**

  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 already permuted by the permutation vectors in *PERM* and *INVP* and it must be used as is by *WSSMP*.

  IPARM(8) cannot be 1 on input if ordering is one of the steps being performed.

  It is valid for *IPARM(8)* to have a value of 0 during ordering and/or 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 in the parallel case. However, the number of rows/columns per process (N_i for Process i) should remain the same for all phases.

- **IPARM(9) or iparm[8], type I:**

  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(8) cannot be 1 on input if ordering is one of the steps being performed.

- **IPARM(10) or iparm[9], type I:**

  The input in *IPARM(10)* determines whether or not a scaling of the input matrix and vector(s) will be performed.

  If *IPARM(10)* = 1, then *WSSMP* performs a scaling that is generally appropriate for LL^T factorization or LDL^T factorization without pivoting. If *IPARM(10)* = 2, then no scaling is performed. If *IPARM(10)* = 3, then *WSSMP* performs a scaling that is generally appropriate for LDL^T factorization with pivoting.

  If *IPARM(10)* = 0, which is the default, then the scaling that is generally best for LDL^T factorization with pivoting is performed if *IPARM(31)* is 2 or 4, otherwise, scaling is not performed.

  Note that for factorization without pivoting, the valid inputs in *IPARM(10)* are 0, 1, and 2. For LDL^T factorization with pivoting, the valid inputs in *IPARM(10)* are 0, 1, 2, and 3. Although the type of ordering and scaling performed with *IPARM(10)* = 3 is usually more suitable for indefinite systems that require pivoting, sometimes, the simpler ordering and scaling performed when *IPARM(10)* = 1 suffices and saves some preprocessing time. Therefore, for LDL^T factorization with pivoting, the users are encouraged to experiment with *IPARM(10)* = 1 and 3, and choose the one that works best for their application.

  IPARM(10) is accessed at the time of ordering and it is important to set it to the desired value before ordering. For LDL^T factorization with pivoting, the array *AVALS* must contain valid values at the time of ordering.

  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 is either already permuted or is permuted on output; i.e., if *IPARM(4)* is 0 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.

- **IPARM(11) or iparm[10], type I:**

  IPARM(11) instructs *WSSMP* and *PWSSMP* how to handle very small, zero, or negative (in case of LL^T factorization) pivots.

  We first describe the role of *IPARM(11)* in LL^T factorization. If *IPARM(11)* = 0, then if *WSSMP* encounters a diagonal value less than or equal to *DPARM(10)* 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 or equal to *DPARM(10)* is encountered, then it is replaced by the value in *DPARM(21)* 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)*, *MRP(I)* is set to the integer

value in *IPARM(13)* to flag the occurrence of a bad I-th pivot. The number of occurrences of diagonal values less than or equal to *DPARM(10)* is reported in *IPARM(21)* on output.

The slight difference in the handling of bad pivots in case of LDL^T factorization without pivoting is as follows. In this case, the absolute value of the pivot is compared with *DPARM(10)* and corrective action taken as desired. If *IPARM(11)* is 1 and the absolute value of the pivot is less than or equal to *DPARM(10)*, then it replaced by *DPARM(21)* with the sign of the original pivot. If *IPARM(11)* is 2 and the coefficient matrix is real, then, in addition to the corrective action described above, *MRP(I)* is set to *IPARM(13)* if the I-th pivot was less than or equal to *DPARM(10)* during factorization. Note that both *DPARM(10)* and *DPARM(21)* must always be non-negative.

For LDL^T factorization of a real matrix with pivoting, the role of *IPARM(11)* is as follows. If *IPARM(11)* is 0 and a row/column with all entries smaller than or equal to *DPARM(10)* is encountered during factorization, then *IPARM(64)* is set to the index of this row/column, thus returning with an error condition indicating that the matrix is singular. If *IPARM(11)* is 1, then the factorization proceeds by replacing the diagonal entry by *DPARM(21)* of a column all whose entries are found to be less than or equal to *DPARM(10)* during factorization. If *IPARM(11)* is 2, then in addition to replacing the diagonal with *DPARM(21)*, *WSSMP* sets the entries in *MRP* that correspond to the indices of rows and columns whose diagonal was replaced by *DPARM(21)* to negative integers. In other words, *MRP* marks the set of rows and columns of the matrix that are linearly dependent on a subset of the remaining rows and columns. Upon return, the $N - M$ entries in *MRP* corresponding to a set of linearly dependent rows and columns would contain a negative integer, while the other entries in MRP would be filled with non-negative integers.

Note that *IPARM(11)* = 2 is a valid input for real matrices only. For complex matrices, *MRP* is not accessed and 0 and 1 are the only valid inputs for *IPARM(11)*.

The default value of *IPARM(11)* is 0.

- **IPARM(12) or iparm[11], type I:**

  For LL^T factorization or LDL^T factorization without pivoting, *IPARM(12)* is an input parameter. *IPARM(12)* is not used and is ignored when LDL^T factorization is performed with diagonal pivoting. If *IPARM(12)* = 0, then *DPARM(12)* and *DPARM(22)* are ignored. The default value of *IPARM(12)* is 0.

  An input of *IPARM(12)* = 1 means that in LL^T factorization, if a pivot is greater than *DPARM(10)* but less than or equal to *DPARM(12)*, it will be replaced by *DPARM(22)* during factorization. If *DPARM(12)* $<$ *DPARM(10)* and *IPARM(12)* = 1, then *DPARM(12)* is set to *DPARM(10)*.

  Once again, in the case of LDL^T factorization, the absolute value of the pivot is compared against *DPARM(10)* and *DPARM(12)*. If the pivot is to be replaced, the new pivot has the sign of the old pivot and the value *DPARM(22)*. Just like *DPARM(10)* and *DPARM(21)*, *DPARM(12)* and *DPARM(22)* must always be non-negative.

- **IPARM(13) or iparm[12], type I/O:**

  In factorization without pivoting, *IPARM(13)* is the integer flag used for marking bad pivots in the *MRP* array if *IPARM(11)* = 2.

  When LDL^T factorization is performed with limited pivoting (i.e., when *IPARM(31)* is 6 and *DPARM(11)* is greater than 0.0) then *IPARM(13)* returns the number of diagonal entries that were perturbed in an attempt to keep the factorization numerically stable.

- **IPARM(14) or iparm[13], type I:**

  This option can be used to reuse the space in the arrays *AVALS* and/or *JA* if the user is interested in reducing memory usage and does not need to use either or both of these arrays after factorization. In the distributed-memory parallel version, *IPARM(14)* is ignored in the peer-mode; it is only effective in the 0-master mode. Please refer to Section 8.1 for a description of distributed modes.

Note 5.9 *In order to be effective,* IPARM(14) *must be set before symbolic factorization is performed.*

IPARM(14) = 0 is the default and has no effect. If the amount of memory is not expected to be a constraint, it is best to leave *IPARM(14)* equal to 0 to save copying time. *IPARM(14)* = 0 guarantees that *AVALS* and *JA* will be returned intact.

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. In *PWSSMP*, *IPARM(14)* = 1 triggers a slightly more expensive but less memory intensive method for permuting the matrix for factorization.

If *IPARM(14)* is 2, then *WSSMP* and *PWSSMP* use 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.

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

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.

Note 5.10 *The user must realize that the internal copy of the indices is used in all subsequent calls to factorization and iterative refinement. Therefore,* JA *cannot be reused if the multiple factorizations with same indices and different values are being performed unless the user is supplying a permuted copy of the matrix to factorization; i.e.,* IPARM(8) *= 1 for factorization. Also,* JA *or* AVALS *or both (whichever is being reused) must be passed unaltered to* WSSMP *or* PWSSMP *for iterative refinement steps if these are performed separately.*

- **IPARM(15) or iparm[14], type I:**

  IPARM(15) contains the input parameter *N1*, where $0 \leq N1 \leq N$. *N1* is used only during the ordering phase 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.

  This option is useful for ordering indefinite systems that have a few zero or near-zero values on the diagonal of the coefficient matrix. For such matrices, LDL^T factorization can be performed without pivoting. To ensure the successful completion of the LDL^T algorithm without pivoting, the rows and columns with zero diagonal entries must be placed at the end of the matrix. By ordering these $N - N1$ rows and columns in the end, the user ensures (unless there is numerical cancellation) that these diagonal entries have become nonzero by the time they are factored.

  Another scenario in which it may be useful to use *IPARM(15)* is when there are only a few entries of interest in the RHS vector and the solution. In this case, the entire factor does not need to be saved and the computation in the solve phase can be considerably reduced if the rows and columns of the coefficient matrix corresponding to the entries of interest in the RHS and solution are placed at the end of the matrix. *IPARM(15)* must be set to *N1* if only a subset of the last $N - N1$ entries in the RHS have useful values (all other entries must be 0.0) and the solution corresponding to only these entries is needed. Please refer to the description of *IPARM(34)* for more details.

  IPARM(15) is ignored when LDL^T factorization is performed with diagonal pivoting.

- **IPARM(16) or iparm[15], type I:**

  IPARM(16:20) control the ordering or the generation of the fill-reducing and load-balancing permutation vectors *PERM* and *INVP*.

  If *IPARM(16)* is -1, the ordering is not performed and the original ordering of rows and columns is used. If *IPARM(16)* is -2, then reverse Cuthill-McKee ordering [6] is performed. If *IPARM(16)* is a nonnegative integer, then a graph-partitioning based ordering [8] is performed.

  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)*. *IPARM(16)* = 1 results in the slowest but best ordering, *IPARM(16)* = 3 results in fastest but worst ordering, and *IPARM(16)* = 2 results in an intermediate speed and quality of ordering.

The default value of *IPARM(16)* is 1. When performing only one or a few factorizations per ordering step, it is advisable to change *IPARM(16)* to 3 or 2.

- **IPARM(17) or iparm[16], type I:**

  WSMP uses graph-partitioning based ordering algorithms [8] to minimize fill during factorization. *IPARM(17)* specifies the minimum 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) or iparm[17], type I:**

  A value of 0 in *IPARM(18)* has no effect. The default *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 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 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.

  Note that in the message-passing parallel routine *PWSSMP*, *IPARM(18)* is ignored and the minimum local fill ordering is not performed because it may hamper parallelism in factorization.

- **IPARM(19) or iparm[18], 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) or iparm[19], type I:**

  The input *IPARM(20)* lets the user communicate some known characteristics of the sparse matrix to *WSSMP* 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 categories below, then the default value 0 should be used.

  Certain sparse matrices have a very irregular structure and have a few rows/columns that are much denser than most of the rows/columns. Many sparse matrices arising from linear programming problems fall in this category. For such matrices, the quality and the speed of ordering can usually be improved by setting *IPARM(20)* to 1. This instructs the ordering routine 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 should be used in *IPARM(20)*. This instructs *WSSMP* 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.

  The symbolic factorization phase may fail for some matrices with very irregular structure, unless *IPARM(20)* is set to 1. Note that, if *IPARM(20)* = 0 produces a better ordering for such a matrix, the failure of symbolic factorization can still be avoided by using *IPARM(20)* = 0 during ordering and *IPARM(20)* = 1 during symbolic factorization.

Note 5.11 Recommended options for the serial/multithreaded version: *The recommended contents of* 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* 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* IPARM(16).

Note 5.12 Recommended options for the message-passing 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 processes being used. The ordering speed can be further increased by using a higher value (2 or 3) in* IPARM(16).

Note 5.13 Error code -700 **returned from symbolic factorization:** *If an error code of* -700 *is generated during symbolic factorization, then it can often be corrected by setting* IPARM(20) = *1 before ordering. In other words, the error was probably caused because your matrix is not a regular finite-element type matrix and generates more than expected fill-in for its size.*

Please refer to the description of IPARM(64) *for more details on error code* -700 *if setting* IPARM(20) = *1 does not fix it.*

- **IPARM(21) or iparm[20], type O:**

  Please refer to the description of *IPARM(11)*. If *IPARM(11)* is set to 1 or 2 and *DPARM(10)* and *DPARM(21)* contain their default values, then *IPARM(21)* returns $N - M$, where M is the rank of the $N \times N$ input matrix. In determining the rank of a matrix, *DPARM(10)* is used as the singularity threshold. By setting *DPARM(21)* to a large value, such as the default 10^{200}, the user can obtain a solution even when the coefficient matrix singular. The solution will contain zeros in the $N - M$ locations corresponding to the rows and columns that are found to be linearly depending on the remaining M rows and columns of the matrix.

  Note that if *DPARM(10)* and *DPARM(21)* do not contain their default values as shown in Table 1, then output in *IPARM(21)* may not have any relation with the rank of the matrix. It will return the number of diagonal pivots replaced by *DPARM(21)*.

- **IPARM(22) or iparm[21], type O:**

  IPARM(22) returns the total number of negative eigenvalues of the coefficient matrix. In the case of LDL^T factorization with pivoting, it is the sum of the number of 2×2 diagonal blocks and the number of negative 1×1 diagonals. For factorization without pivoting, it is the number of negative diagonal entries.

- **IPARM(23) or iparm[22], type O:**

  The output *IPARM(23)* is set after the symbolic factorization phase. It contains the total number of Kilo words of memory (8-byte double precision words) that *WSSMP* or *PWSSMP* will require for factorization. This includes working storage as well as the space to store the factor. For LDL^T factorization with diagonal pivoting, *IPARM(23)* is updated after numerical factorization to reflect the actual memory used by the factors, because this may be larger than that predicted by symbolic factorization due to pivoting.

  Note that, due to round-off errors, the value of *IPARM(23)* may not be very accurate for very small matrices.

- **IPARM(24) or iparm[23], type O:**

  On output, *IPARM(24)* contains the number of nonzeros in the triangular factor in thousands. This field is set after the symbolic factorization phase.

  Note that, due to round-off errors, the value of *IPARM(24)* may not be very accurate for very small matrices.

- **IPARM(25)^{S,T} or iparm[24], type I:**

  IPARM(25) is ignored when *IPARM(31)* is less than 5. For *IPARM(31)* = 5, 6, or 7, 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 [4] 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.

  The computation of $\|A\|$ returned in *DPARM(2)* is straightforward. The estimate for $\|A^{-1}\|$ returned in *DPARM(3)* is computed as follows. Let $A = LL^T$ be the factorization. Then a system $Lz = d$ is solved, where the elements of vector d are chosen from 1.0, -1.0 such that $\|z\|$ is maximized. Then $L^T y = z$ is solved, followed by $Lw = y$ and $L^T x = w$. The estimate for $\|A^{-1}\|$ is $\|x\|/\|y\|$. The process is similar in the case of LDL^T factorization, where D also participates in the computation.

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

  Condition number estimation is not yet available in the message-passing parallel version. It is available for factorization without pivoting and with static memory allocation in serial/multithreaded mode only; i.e., when *IPARM(31)* = 5, 6, or 7.

  The default value of *IPARM(25)* is 0; i.e., condition number estimation is not performed.

- **IPARM(26)^{P} or iparm[25], type I or M:**

  This parameter is relevant only in the message-passing 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)^{T,P} or iparm[26], type I:**

  This parameter, which is relevant in the message-passing and the multithreaded versions, allows the user to control the load-balance versus fill-in trade-off to some extent.

  IPARM(27) = 2 allows the fill-reducing ordering to determine load-balancing among the processes with only slight adjustments.

  If *IPARM(27)* = 1, then *WSSMP* and *PWSSMP* perform 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). So this option should be used with discretion and aggressive load balancing should be turned on by setting *IPARM(27)* to 1 only when poor speedups are observed and load-imbalance is suspected to be the culprit. Our experience has shown that using *IPARM(27)* = 1 is significantly beneficial for factoring LP (Linear Programming) and other highly unstructured matrices. In very rare cases, using *IPARM(27)* = 1 for matrices arising from regular grids may cause symbolic factorization to fail.

  The default value of *IPARM(27)* is 0 allows *WSSMP* and *PWSSMP* to decide what type of restructuring of the elimination tree to perform.

- **IPARM(28)^{P} or iparm[27], type I:**

IPARM(31)	Factorization	Memory allocation	Pivoting	Matrix type
0	LL^T	Dynamic	No	Real/Hermitian
1	LDL^T	Dynamic	No	Real/Hermitian
2	LDL^T	Dynamic	Yes^{S,T}/Limited^P	Real/Hermitian
3	LDL^T	Dynamic	No	Complex non-Hermitian
4	LDL^T	Dynamic	Yes^{S,T}/Limited^P	Complex non-Hermitian
5	LL^T	Static	No	Real/Hermitian
6	LDL^T	Static	Limited	Real/Hermitian
7	LDL^T	Static	Limited	Complex non-Hermitian

Table 2: The first column contains the possible input values for *IPARM(31)*, the second column shows the type of symmetric factorization performed, the third column shows the type of memory allocation performed, the fourth column indicates whether or not pivoting is performed, and the last column shows the type of input coefficient matrix. Superscript ^{S} denotes *serial*, ^{T} denotes *multithreaded*, and ^{P} denotes *MPI parallel*

.

This parameter is relevant only in the message-passing 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) or iparm[28], type I:**

  If LDL^T factorization with diagonal pivoting is performed, *WSSMP* 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. If *IPARM(29)* is 0, this extra memory is not reclaimed. By default, *IPARM(29)* is 0, because usually the amount of extra memory is small and is not worth the overhead of reclaiming it. If *IPARM(29)* is set to 1, then *WSSMP* performs garbage collection to remove memory holes if it runs short of memory during factorization. If *IPARM(29)* is set to 2, then *WSSMP* always performs a garbage collection step at the end of factorization and returns with only as much memory allocated as needed to store the factors.

- **IPARM(30) or iparm[29], type I:**

  IPARM(30) is read during the triangular solution phase, which performs both forward and backward solution if *IPARM(30)* is set to its default value of 0. If *IPARM(30)* is 1, then only the forward (lower triangular) solution is performed. If *IPARM(30)* is 2, then only the backward (upper triangular) solution is performed. For LL^T factorization, calling the solve phase of *WSSMP* or *PWSSMP* with *IPARM(30)* = 0 should produce the same output in B as a call with *IPARM(30)* = 1 followed by a call with *IPARM(30)* = 2. However, if forward and back solves are performed separately, then the user looses the ability to perform iterative refinement on the solution thus obtained.

  In case of LDL^T factorization, an additional diagonal solution step is required between the forward and the backward steps. In this case, if *IPARM(30)* is 0, then all three steps are performed, if *IPARM(30)* is 3 then only the diagonal step is performed, if *IPARM(30)* is 4 then backward and diagonal steps are performed, and if *IPARM(30)* is 5, then forward and diagonal steps are performed. Input values of 1 and 2 for *IPARM(30)* result in only forward and backward triangular solutions, respectively.

- **IPARM(31) or iparm[30], type I:**

The input *IPARM(31)* can be used to select the appropriate algorithm depending on the characteristics and type of the coefficient matrix. Table 2 shows the eight possible input values for *IPRM(31)* and the corresponding factorization algorithm and matrix type. The default value of *IPARM(31)* is 0.

If the matrix is known to be positive-definite, then LL^T factorization is recommended as it is somewhat faster than LDL^T factorization in the current implementation. Similarly, if LDL^T factorization is known to be stable without pivoting, then pivoting should be avoided because pivoting slows down the factorization and solve phases. For solving symmetric indefinite systems that require pivoting, WSMP provides two pivoting options. Both options use 1×1 and 2×2 pivot blocks as in the algorithm by Bunch and Kaufman [2]. In one option, which is available only in serial and multithreaded modes, pivots can be selected from anywhere in the matrix to satisfy the pivoting threshold. In the second limited pivoting option, which is useful in the MPI parallel case, pivot search is localized and is not global. Localized pivot search keeps the communication overhead under control, and while usually successful, can sometimes fail to find a suitable pivot. In this situation, a small diagonal perturbation is introduced and make one of the local pivots viable by increasing the magnitude of the corresponding diagonal entry. The accuracy lost as a result of the perturbation is usually recovered via iterative refinement.

WSMP factorizations also offer two memory allocation strategies. Some factorization algorithms allocate temporary working space and the memory for storing the factors dynamically as needed. Some algorithms allocate all the working space and the memory for storing the factor in one large block before the factorization starts. For matrices with sparse factors and slim supernodes, the dynamic memory algorithms can incur a significant overhead relative to the total amount of computation and may be slower than the static allocation algorithm. However, dynamic allocation results in a smaller overall memory footprint because the memory for storing the entire factor has not been allocated at the time of peak temporary working storage demand. Table 2 shows the memory allocation scheme corresponding to each value of *IPARM(31)*.

Only 0, 1, 2, 5, and 6 can be used as inputs in *IPARM(31)* for real matrices. For complex matrices (Section 11), input values of 3, 4, and 7 should be used to indicate that the matrix is non-Hermitian, and the remaining values can used for Hermitian matrices.

Note 5.14 *The input in* IPARM(31) *is read during the ordering phase; therefore, it is important that the* IPARM(31) *contain the proper choice of factorization method to be applied before ordering. Switching between LL^T and LDL^T factorization without pivoting is permitted at the time of factorization.*

- **IPARM(32) or iparm[31], type I:**

The default value of *IPARM(32)* is 0. By setting *IPARM(32)* to a suitable value, the user can request certain output in *DIAG*. If *IPARM(32)* is greater than 0 on input, then the user must provide a double precision array of length at least N in *DIAG*.

WSSMP does not give user the access to the factors of the coefficient matrix, which is stored in a non-standard format in internally allocated data structures. However, a user may want to inspect the diagonal values of the factor to analyze the numerical accuracy of the factorization process. If *IPARM(32)* is 1 on input, then the diagonal of the Cholesky factor is copied into *DIAG* (provided that task number 3 is performed during that call to *WSSMP*).

If the option of getting the diagonal of the factor back in *DIAG* is used with LDL^T factorization (without pivoting), then the output actually contains the inverse of the diagonal D.

If *IPARM(32)* is set to 2 on input, then the vector y, as described for *IPARM(25)*, computed during the estimation of $\|A^{-1}\|$ is returned in *DIAG*. See description of *IPARM(25)* for more details. Note that the user must activate the condition number estimation by setting *IPARM(25)* to a valid value greater than 0 in order to utilize this option of getting y back in *DIAG*. Note that since the condition number estimation works on the scaled matrix if *IPARM(10)* is greater than 0 and the vector d as described for *IPARM(25)* is chosen on the fly, scaling must be turned off for the *IPARM(32)* = 2 option to work correctly.

Note 5.15 IPARM(32) *= 2 is supported only if* IPARM(31) *is 5, 6, or 7 because condition number estimate is computed for only these values of* IPARM(31).

- **IPARM(33) or iparm[32], type O:**

  On output, *IPARM(33)* is set to the number of CPU's that were used by the process in SMP mode. Please refer to Section 3.4 for details on controlling the number of threads in *WSMP*.

  In *PWSSMP*, the output in *IPARM(33)* is local to each MPI process.

- **IPARM(34) or iparm[33], type I:**

  If *IPARM(34)* is set to $N1$ upon input, then only the last $N - N1$ rows of B are assumed to contain nonzero values upon input and are considered to be relevant for output. The input *IPARM(34)* is read at solve time and its use requires that the first $N1$ entries of B are set to 0.

  For *IPARM(34)* to be effective, *IPARM(15)* must be used in the ordering phase and *IPARM(34)* must be greater than or equal to the value used in *IPARM(15)* during ordering.

  Note that if *N1* is much smaller than N, then it may be inefficient to obtain a partial solution and the user may be better off using the normal solution process and just using the relevant parts of the solution. The default value of *IPARM(34)* is 0, which has no effect on the solution.

  Note that if the backward and forward solution steps are performed separately, then *IPARM(34)* can be applied selectively to either step.

- **IPARM(35)^P or iparm[34], type I:**

  This parameter is relevant only for the message-passing parallel version and has a default value of 0. If *IPARM(29)* is 1, *PWSSMP* attempts to distribute the Cholesky factorization work according to the speed and load of different processes on a heterogeneous machine. An input value of 0 switches off this optimization and guarantees the same *PERM* and *INVP* for different runs on the same problem with the same number of processes, provided that the input parameters for ordering, *IPARM(16:20)*, are the same. Note that this parameter is read during symbolic factorization and must be set to the appropriate value before this step.

- **IPARM(36)^{S,T} or iparm[35], type M:**

  IPARM(36) can be used to enable out-of-core (OOC) computation by setting it to 1 or 2. Its value is 0 by default; i.e., OOC computation is turned off. OOC computations can only be used with *IPARM(31)* values of 0 or 1. Please refer to Section 6 for more details on OOC computations, including the setting of mandatory environment variables.

- **IPARM(37:63) or iparm[36:62], type R:**

  These are reserved for future use.

- **IPARM(64) or iparm[63], type O:**

  In the event of a successful return from *WSSMP* or *PWSSMP*, *IPARM(64)* is set to 0 on output. A nonzero value of *IPARM(64)* upon output is an error code, and indicates that *WSSMP/PWSSMP* did not complete execution and detected an error condition. There are two types of error codes—negative and positive. In *PWSSMP*, the error code returned on all MPI processes is identical. The three least significant decimal digits indicate the error code and the remaining most significant digits indicate the MPI process number that was the first to encounter the error. For example, an error code of -102 indicates that process 0 detected error -102 and an error code of -2700 indicates that process 2 detected error -700. The value of *IPARM(64)* will be set to -102 and -2700, respectively, upon return on all the processes.

  Negative Error Codes: A two-digit negative error code indicates an invalid input argument. 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.

A three-digit negative error code indicates a non-numerical run-time error.

If dynamic memory allocation by *WSSMP* fails, then *IPARM(64)* is set to -102 on return. This is one of the most common error codes encountered by the users. Please refer to Section 3.1 if you get this error in your program.

An error code of -103 is sometimes generated for very large matrices if the software encounters an integer overflow. Factorization algorithms with static memory allocation (i.e., when *IPARM(31)* is 5, 6, or 7) are particularly prone to this error when the size of the factor matrix exceeds 2^{31}. On many platforms, a special library *libwsmp8_8.a* is available. This library uses 8-byte integers and will solve the problem. Please make sure that all integer parameters that are passed to *WSMP* routines are of type *integer*8* in Fortran or *long long* in C (either declared explicitly, or by using the appropriate compiler option to promote all integers to 8-byte size) when using *libwsmp8_8.a*.

An output value of -200 in *IPARM(64)* in the message-passing parallel version indicates that the problem is too small for the given number of processes and must be attempted on fewer processes. The -200 error code is also returned if MPI is not initialized before a call to a *PWSMP* routine.

An error code of -300 is returned if the current operation is invalid because it depends on the successful completion of another operation, which failed or was not performed by the user. For example, if Cholesky factorization fails and you call *WSMP* to perform backsolves after the failed call for factorization, you can expect error -300.

An output value of -700 in *IPARM(64)* indicates an internal error and should be reported to *wsmp@us.ibm.com*. If an error code of -700 is generated during symbolic factorization, then it can often be corrected by setting *IPARM(20)* = 1 before ordering. In other words, the error was probably caused because the matrix is not a regular finite-element type matrix and generates more than expected fill-in for its size.

An error code of -900 is returned if the license is expired, invalid, or missing.

In addition to the above error codes, there are some other negative error codes that can result when out-of-core computations are used. These are described in Section 6.3.

Positive Error Codes: A positive integer value of *IPARM(64)* between 1 and N on output indicates that the coefficient matrix is singular or close enough to singular that factorization cannot proceed beyond *IPARM(64)* rows and columns. For LL^T factorization, *IPARM(64)* is the index of the first pivot that was less than or equal to *DPARM(10)* before computing the square root for Cholesky factorization. For LDL^T factorization without pivoting, it is the index of the first pivot whose absolute value is less than or equal to *DPARM(10)*. Note that if C-style (0-based) indexing is used and *IPARM(64)* > 0, then *IPARM(64)* is 1 + the index of the bad pivot.

For LDL^T factorization with pivoting, a positive error code is returned in *IPARM(64)* only when *IPARM(11)* is set to 0. This denotes that a singularity was encountered. Unlike the factorization without pivoting where the positive *IPARM(64)* value points to the location (row/column index) of singularity in the coefficient matrix, for LDL^T factorization with pivoting, the actual positive value returned in *IPARM(64)* is meaningless.

Note 5.16 *Note that in case of an out-of-memory error in the distributed-memory parallel solver, one or more of the input data arrays may be corrupted.*

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

```
DOUBLE PRECISION DPARM ( 64 )
double dparm[]
```

The entries *DPARM(36)* through *DPARM(63)* are reserved. Unlike *IPARM*, only a few of the first 35 entries of *DPARM* are used. The description of only the relevant entries of *DPARM* is given below. Note that all reserved entries; i.e., *DPARM(36:63)* must contain 0.0.

- **DPARM(1) or dparm[0], type O:**

  Returns the total wall clock time in seconds spent in an *WSSMP* or *PWSSMP* call. Since this is the elapsed time, it can vary depending on the load on the machine and several other factors.

- **DPARM(2) or dparm[1], type O:**

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

- **DPARM(3) or dparm[2], type O:**

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

- **DPARM(4) or dparm[3], type O:**

  On output, this contains the largest diagonal element encountered in the factorization process (just before the square root in LL^T factorization). For LDL^T factorization, *DPARM(4)* is the diagonal element with the greatest magnitude. 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 is replaced by a large value, *DPARM(4)* = $\max_{i=1,N} L(i,i)^2$ for LL^T factorization and *DPARM(4)* = $D(i)$ such that $D(i) = \max_{i=1,N} |D(i)|$ for LDL^T factorization.

- **DPARM(5) or dparm[4], type O:**

  On output, this contains the smallest diagonal element encountered in the factorization process (just before the square root in LL^T factorization). For LDL^T factorization, *DPARM(5)* is the diagonal element with the smallest magnitude. The minimum 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 being replaced, *DPARM(5)* = $\min_{i=1,N} L(i,i)^2$ for LL^T factorization and *DPARM(5)* = $D(i)$ such that $D(i) = \min_{i=1,N} |D(i)|$ for LDL^T factorization.

- **DPARM(6) or dparm[5], 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 relative residual norm becomes less than *DPARM(6)*. *DPARM(6)* is not used if *IPARM(7)* = 0.

- **DPARM(7) or dparm[6], type O:**

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

  In the message-passing parallel version, the relative residual norm is not computed and iterative refinement is not performed if *NRHS* > 20. So if more than 20 solutions are required with iterative refinement or relative residual norm computation, then these must be performed in batches of at most 20 each.

- **DPARM(10) or dparm[9], type I:**

  For factorization without diagonal pivoting, the input in *DPARM(10)* serves as the lower threshold on value of a good diagonal value. If a pivot value is less than or equal to *DPARM(10)*, then either an error is flagged or corrective action is taken as specified by *IPARM(11)*. *DPARM(10)* must be non-negative. In case of LDL^T factorization without pivoting, *DPARM(10)* is treated as the lower threshold on the absolute value of a good pivot. Please refer to the description of *IPARM(11)* for more details.

  For LDL^T factorization with pivoting, the input in *DPARM(10)* is used as the threshold for determining if a matrix is singular. If a leading row or column is encountered in the unfactored part of the matrix such that all its entries are less than or equal to *DPARM(10)*, then the matrix is deemed singular and this condition is reported in *IPARM(64)*. The default value of *DPARM(10)* is 10^{-18}. The default value of *DPARM(10)* is appropriate only if the matrix is scaled. If the matrix is not scaled, then the user must specify an appropriate threshold in *DPARM(10)* to detect singularity.

  DPARM(10) is accessed at the time of ordering and it is important to set it to the desired value before ordering.

- **DPARM(11) or dparm[10], type I:**

  For LDL^T factorization with diagonal pivoting, *DPARM(11)* is the pivoting threshold used in the Bunch-Kaufman [2] algorithm. A value greater than 0.64 for *DPARM(11)* is not recommended, and in practice, much smaller values usually work quite well. A smaller value of *DPARM(11)* reduces the time and memory requirement of factorization, but also reduces the numerical stability and the accuracy of the solution. Often, some or most of this loss of accuracy can be recovered with a few steps of iterative refinement. The default value of *DPARM(11)* is 0.001.

  DPARM(11) is accessed at the time of ordering and it is important to set it to the desired value before ordering. *DPARM(11)* is not accessed when either LL^T factorization or LDL^T factorization without pivoting are chosen.

- **DPARM(12) or dparm[11], type M:**

  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(12)* is ignored if LDL^T factorization is performed with pivoting.

- **DPARM(13) or dparm[12], type O:**

  After symbolic factorization, *DPARM(13)* contains the number of supernodes detected. A small number of supernodes relative to the size of the coefficient matrix indicates larger supernodes and hence, higher potential performance in the numerical steps.

- **DPARM(15) or dparm[14], type M:**

  DPARM(15) is accessed by *WSSMP* during the numerical LDL^T factorization phase with pivoting; i.e., when *IPARM(31)* is 2 or 3. Typically, for a sequence of matrices with the same structure but changing values, ordering and symbolic factorization steps are performed only once, followed by repeated factorization and solve steps. For LDL^T factorization with pivoting, however, ordering is partly based on the numerical values in the coefficient matrix. As the coefficient matrix values change, the ordering may become outdated and may need to be recomputed for optimum overall performance. If *DPARM(15)* is set to 1.0 while refactoring a matrix (i.e., *IPARM(2)* is 3) for which ordering and symbolic factorization has already been computed, then the ordering is refreshed based on the current values in the coefficient matrix.

  The default value of *DPARM(15)* is 0.0 and when an input of 1.0 is supplied to trigger an update of the ordering for the new values, it is reset to 0.0 on output.

  Refreshing the ordering can be costly and must be performed judiciously. Refreshing the ordering too frequently can degrade the overall performance. In many applications, the ordering never needs to be refreshed. The use of *DPARM(15)* is recommended for advanced user only. Typically, determining when to refresh the ordering requires careful tracking of factorization memory or time and solution error. Alternatively, a user may experiment with fixed intervals for refreshing the ordering (say, every 10 or 20 factorization steps) and choose the best for their application.

  Note that, while using *IPARM(2) = 3* and *DPARM(15) = 1.0* is semantically equivalent to using *IPARM(2) = 1* and *DPARM(15) = 0.0*, the first option is faster and involves less recomputation than the second option.

- **DPARM(21) or dparm[20], type I:**

  Please refer to the description of *IPARM(11)*. the default value of *DPARM(21)* is 10^{200}. *DPARM(21)* must be non-negative.

  DPARM(21) is ignored if LDL^T factorization is performed with pivoting.

- **DPARM(22) or dparm[21], type I:**

  Please refer to the description of *IPARM(12)*. If *DPARM(22)* is being used; i.e., if *IPARM(12)* = 1, then it is the user's responsibility to initialize *DPARM(22)* with a valid positive double precision value. *DPARM(22)* must be non-negative.

DPARM(22) is ignored if LDL^T factorization is performed with pivoting.

- **DPARM(23) or dparm[22], type O:**

  After the symbolic factorization phase, *DPARM(23)* is the same *DPARM(24)*; i.e., it contains the expected number of floating point operations required for factorization. After the numerical factorization phase, *DPARM(23)* is updated to contain the actual number of floating point operations performed during factorization. For factorization without pivoting, this is the same as *DPARM(24)*, but is usually higher for LDL^T factorization with diagonal pivoting.

- **DPARM(24) or dparm[23], type O:**

  After the symbolic factorization phase, *DPARM(24)* contains the expected number of floating point operations required for factorization.

- **DPARM(64) or dparm[63], 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)*.

6 The Out-of-Core Solver: Using Secondary Storage

WSMP's Cholesky and LDL' factorization functions (and the corresponding solve phases) without pivoting can be performed out-of-core using secondary storage for matrices whose factorization requires more memory than what is available to the process. The out-of-core (OOC) functionality can be used only if *IPARM(31)* is 0 or 1, and is not available in distributed memory. The OOC functionality is also not available through the simple interface described in Section 7. In other words, OOC functionality can be used with the *WSSMP* routine (Section 5) only, and only if *IPARM(31)* is 0 or 1.

The OOC functionality can be triggered by setting *IPARM(36)* to 1 or 2. If *IPARM(36)* is 0, then an in-core factorization is attempted, which could result in error -102 in *IPARM(64)* upon return if memory is insufficient. If *IPARM(36)* is 1, then an OOC factorization is attempted. If *IPARM(36)* is 2, then *WSSMP* decides whether to perform the factorization in-core or out-of-core, based on the amount of in-core memory available and the expected size of the factors. If *IPARM(36)* is 2, then it is set to 0 or 1 after the symbolic factorization phase to indicate whether the numerical factorization will be performed in-core or out-of-core.

In addition to *IPARM(36)*, two mandatory environment variables, namely *WOOCDIR0* and *WINCOREMEM* must be set for *WSSMP* to perform OOC factorization. These, and some other optional environment variables related to OOC factorization are summarized in Section 6.1 below. In addition to these environment variables, it is extremely important to set certain system parameters appropriately for the OOC functionality of *WSMP* to work effectively. Setting these may require *root* privileges. These are described in Section 6.2.

6.1 Environment Variables for OOC Computation

1. *WOOCDIR0:*

   WSSMP needs to know the path to the directories in which it can store the factor and temporary arrays during factorization. If *IPARM(36)* is nonzero, then *WOOCDIR0* is a mandatory environment variable that must be set to the location where *WSSMP* can create temporary files when running in out-of-core mode. If *WOOCDIR0* is not set, or if *WSSMP* is unable to open files in the directory specified by *WOOCDIR0*, then error -502 is returned in *IPARM(64)*.

   WSSMP can use up to 8 different directories among which to distribute the temporary files. This can be useful if a single location does not have a free space. Moreover, if the machine has multiple secondary storage devices, then using all of them may improve performance. Additional locations are specified by setting environment variables *WOOCDIR1*, *WOOCDIR2*, ..., *WOOCDIR7*. Note that *WOOCDIR0* is mandatory, but *WOOCDIRk*, $1 \leq k \leq 7$

are optional. The first $k + 1$ locations are used if *WOOCDIR0* ... *WOOCDIRk* are set for all m, $0 \leq m \leq k$ and $k < 8$.

2. *WINCOREMEM:*

   This is another environment variable that is mandatory if *IPARM(36)* is nonzero. *WINCOREMEM* is used to indicate to *WSSMP* roughly how much in-core memory in megabytes it is allowed to use. If the sum of all bookkeeping and factorization memory requirement exceeds *WINCOREMEM* megabytes, then *WSSMP* uses secondary storage to supplement the in-core memory.

   Note that while setting *WINCOREMEM*, the user must take into account the total memory available on the machine, whether or not other processes are using part of that memory, the amount of memory used by the user's application outside *WSMP*, and the fact that some memory will inevitably be unused due to fragmentation. While larger in-core memory, in general, improves performance, the user must be careful to not set *WINCOREMEM* to a value that results in paging/swapping. A process that performs automatic paging/swapping will run considerably slower than one in which *WSMP* explicitly orchestrates writes and reads to secondary storage.

   If *IPARM(36)* is nonzero and *WINCOREMEM* is not set, then error -501 will be returned in *IPARM(64)*.

3. *WMAXOOCFILESIZE:*

   This is an optional environment variable that indicates that maximum size, in megabytes, of a single out-of-core file. The default size is 1024 MB.

4. *WMINWRTBLK:*

   This is an optional environment variable that can be used to specify the minimum size, in megabytes, of the block for write operations of factor data to secondary storage. By default, the factor is written to secondary storage in blocks of at least 8 MB (equivalent to WMINWRTBLK set to 8); i.e., write operations with smaller data are consolidated.

5. *WOOCTASKCCSIZE:*

   This is an optional environment variable that can be used to specify the size, in megabytes, of task-private storage of contribution blocks. The default size is 16 MB (equivalent to WOOCTASKCCSIZE set to 16). Please refer to the technical paper [1] on *WSMP*'s OOC solver for more details on task-private storage. The user may try to tune it for the best performance for the application and hardware combination.

6. *WFACTORMINMEM:*

   This is an optional environment variable that can be used to specify the target minimum percentage of in-core memory that will be used to store the factor. If space to store the factor drops below a certain percentage of in-core memory, then *WSMP* employs a technique called "pushing down of parallelism" in an attempt to store more of the factor data in memory [1]. Excessive pushing down of parallelism can reduce concurrency. The default value is 70% (corresponding to a setting of 70 for *WFACTORMINMEM*). The user may try to tune it for the best performance for the application and hardware combination.

7. *WLPREFIX* and *WUPREFIX:*

   WSMP uses the default prefix *wsmp_tmp_scratch* for the files that store factor (L) and update (U) matrices. The user can change the default prefixes for L and U matrices by setting the *WLPREFIX* and *WUPREFIX* environment variables, respectively.

   The purpose of these prefixes is to make these files readily identifiable as temporary files that are useful only while the process that created them is active. In fact, the process id itself is also a part of the file names. *WSMP* deletes the temporary files when these are no longer needed. However, if the process terminates abnormally or is killed by the user, then the temporary files are not deleted. The user will then need to delete the leftover files manually.

6.2 System Settings for OOC Computation

Modern operating systems are generally set up to use main memory as a cache for the secondary storage when possible. To some extent, this is useful for OOC solvers as it prevents excessive disk access, which can be costly. However, beyond a point, this can interfere with the careful management of the primary and secondary storage that *WSMP*'s OOC solver incorporates. One of the reasons is that when some data is scheduled to be written on to secondary storage, the OS generally does not have knowledge of the future use of this data. As a result, it may cache some persistent data and may expel transient data to secondary storage. This can result in a large performance degradation. In order to avoid this performance degradation, certain system settings must be changed from their default values. Changing these settings may require *root* privileges (by logging in as *root* or through *sudo*). We describe the changes for AIX and Linux below; similar changes may be required on other systems as well.

6.2.1 Linux

On Linux systems, the file */proc/sys/vm/swappiness* must be edited to contain a 0. With *root* privilege, this can be accomplished by the following command:
% echo 0 > /proc/sys/vm/swappiness

Note that the user may want to back the original file up to restore swappiness to its original value when the machine is not being used to the run the OOC solver.

6.2.2 AIX

On an AIX system, the user can use the *smitty* utility with *root* privileges to change the value of *minperm* to 3, *maxperm* to 5, *maxclient* to 5, *strict_maxperm* to 1, and *lru_file_repage* to 1. These can be accessed by successively choosing the following from the smitty menu: (1) Performance & Resource Scheduling, (2) Tuning Kernel & Network Parameter, (3) Tuning Virtual Memory Manager, File System and Logical Volume Manager Params, (4) Change / Show Current Parameters.

The user may want to restore these to the original values when the machine is not being used to the run the OOC solver.

6.3 Possible Errors During OOC Computation

In addition to the possible error codes described in Section 5.2.14 that can be returned in *IPARM(64)*, there are some errors that are exclusive to OOC computation. Failure to set environment variables *WINCOREMEM* and *WOOCDIR0*, when *IPARM(36)* is 1 or 2, may result in errors -501 and -502, respectively. Error -503 results due to a failure to write to secondary storage, most likely due to the storage device being full. Error -504 is returned in *IPARM(64)* if the value of *WINCOREMEM* is insufficient for the given coefficient matrix.

7 Subroutines Providing a Simpler Serial/Multithreaded Interface

In this section, we describe a simpler interface to *WSSMP*. This interface accepts the input in both CSR/CSC and MSR/MSC 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 5, unless mentioned otherwise. The *WSMP* home page contains an example driver program *wssmp_ex2.f* that uses the simple interface. The calling sequences of these subroutines are described below.

Note 7.1 *The calls to the* WSSMP/PWSSMP *routines should not be mixed with those to the routines in the simple interface described in this section and in Section 9. The user must choose to use either the* WSSMP/PWSSMP *routines or the simple interface for a given application, and stick to the chosen interface.*

Note 7.2 *The simple interface routines perform factorization with static data allocation. Please refer to the description of* IPARM(31) *and Table 2 for more details. If you wish to perform factorization with dynamic memory allocation, which may be more suitable in certain circumstances, please use the* WSSMP *routine.*

7.1 WMMRB, WKKTORD, WN1ORD (ordering)

WMMRB (N, XADJ, ADJNCY, OPTIONS, NUMBERING, PERM, INVP, AUX, NAUX)
void *wmmrb_(int *n, int *xadj, int *adjncy, int *options, int *numbering, int *perm, int *invp, int *aux, int *naux)*

WKKTORD (N, XADJ, ADJNCY, OPTIONS, NUMBERING, PERM, INVP, AUX, NAUX)
void *wkktord_(int *n, int *xadj, int *adjncy, int *options, int *numbering, int *perm, int *invp, int *aux, int *naux)*

WN1ORD (N, N1, XADJ, ADJNCY, OPTIONS, NUMBERING, PERM, INVP, AUX, NAUX)
void *wn1ord_(int *n, int *n1, int *xadj, int *adjncy, int *options, int *numbering, int *perm, int *invp, int *aux, int *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. The routine *WN1ORD* is useful for ordering indefinite systems that have zero or near-zero values on the diagonal of the coefficient matrix. To ensure the completion of the LDL^T algorithm, the rows and columns with zero diagonal entries must be placed at the end of the matrix. By ordering these $N - N1$ rows and columns in the end, the user ensures (unless there is numerical cancellation) that these diagonal entries have become nonzero by the time they are factored.

7.1.1 N, (type I): matrix dimension

```
INTEGER N
int *n
```

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

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

```
INTEGER XADJ (NUMBERING : N + NUMBERING)
int *xadj
```

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.

7.1.3 ADJNCY, (type I): adjacency list

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

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.

7.1.4 OPTIONS, (type I): ordering options

```
INTEGER OPTIONS(1:5)
int *options
```

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)*. The only exception is *OPTIONS(4)*, which also serves as an output for indicating the status of the call to *WMMRB*. In case of *WSSMP/PWSSMP*, *IPARM(19)* is a pure input argument. On output, a -102 in *OPTIONS(4)* indicates that dynamic memory allocation failed (see Section 3.1), 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.

Note 7.3 *Note that* WSSMP*, by default, passes (1,0,1,0,0) in* OPTIONS *to* WMMRB *(Table 1, Section 5.2). This should not be confused with the fact that the default values of (0,0,0,0) are used for* OPTIONS(2:4) *if* OPTIONS(1) *is 0 in the three ordering routines described in this section.*

Note 7.4 OPTIONS(4) *is used to output an error condition (if one arises) only in a direct call to* WMMRB*. In* WSSMP*,* IPARM(19) *is always unchanged and only* IPARM(64) *is used to flag all errors, including the ones detected in the ordering phase.*

- **OPTIONS(1) or options[0], type I:**

  If *OPTIONS(1)* is -1, the ordering is not performed and the original ordering of rows and columns is returned in *PERM* and *INVP*. If *OPTIONS(1)* is -2, then reverse Cuthill-McKee ordering [6] is performed. If *OPTIONS(1)* is a nonnegative integer, then a graph-partitioning based ordering [8] is performed.

  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) or options[1], type I:**

  By default, *WMMRB* uses graph-partitioning based ordering algorithms [8] to minimize fill during factorization. *OPTIONS(2)* specifies the minimum 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) or options[2], type I:**

  The default value of 0 in *OPTIONS(3)* has no effect. *OPTIONS(3)* = 1 forces *WMMRB* 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.

- **OPTIONS(4) or options[3], 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.

- **OPTIONS(5) or options[4], 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/columns. 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.

7.1.5 NUMBERING, (type I): indexing options

```
INTEGER NUMBERING
int *numbering
```

This is same as *IPARM(11)* described in Section 5.2.14.

7.1.6 PERM, (type O): permutation vector

```
INTEGER PERM ( N )
int *perm
```

The description is the same as in Section 5.2.6.

7.1.7 INVP, (type O): inverse permutation vector

```
INTEGER INVP ( N )
int *invp
```

The description is the same as in Section 5.2.7.

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

```
INTEGER AUX ( NAUX )
int *aux
```

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 unused 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.

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

```
INTEGER NAUX
int *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.

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

WSCSYM (N, IA, JA, PERM, INVP, NNZL, WSPACE, INFO)
void *wscsym_(int *n, int *ia, int *ja, int *perm, int *invp, int *nnzl, int *wspace, int *info)*

WSMSYM (N, IA, JA, PERM, INVP, NNZL, WSPACE, INFO)
void *wsmsym_(int *n, int *ia, int *ja, int *perm, int *invp, int *nnzl, int *wspace, int *info)*

These routines perform symbolic factorization only and uses the input permutation given by *PERM* and *INVP*. On output, the integer *NNZL* contains the number of nonzeros in the Cholesky factor in thousands and the integer *WSPACE* contains the number of *double words* of memory required by Cholesky factorization in thousands, including the storage for the Cholesky factor. 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 5.2.14.

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

WSCALZ (N, IA, JA, OPTIONS, PERM, INVP, NNZL, WSPACE, AUX, NAUX, INFO)
void *wscalz_(int *n, int *ia, int *ja, int *options, int *perm, int *invp, int *nnzl, int *wspace, int *aux, int *naux, int *info)*

void *WSMALZ (N, IA, JA, OPTIONS, PERM, INVP, NNZL, WSPACE, AUX, NAUX, INFO)*
void *wsmalz_(int *n, int *ia, int *ja, int *options, int *perm, int *invp, int *nnzl, int *wspace, int *aux, int *naux, int *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, solve, and iterative refinement routines can be made as long as the nonzero structure of

the coefficient matrices does not change. The description of *OPTIONS*, *AUX*, and *NAUX* inputs is the same as in the calling sequence of *WMMRB* (Section 7.1). 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 thousands in the Cholesky factor and the integer *WSPACE* contains the number of double words of memory required by Cholesky factorization in thousands, including the storage for the Cholesky factor.

The *PERM* and *INVP* outputs of this routine must be passed to all subsequent factorization and solves. This routine needs to be called only once for any number of factorizations and solves, as long as the location of nonzeros in the input matrix remains the same.

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 5.2.14.

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

WSCCHF (N, IA, JA, AVALS, PERM, INVP, INFO)
void *wsschf_(int *n, int *ia, int *ja, double *avals, int *perm, int *invp, int *info)*

WSMCHF (N, IA, JA, AVALS, DIAG, PERM, INVP, INFO)
void *wsmchf_(int *n, int *ia, int *ja, double *avals, double *diag, int *perm, int *invp, int *info)*

These routines perform LL^T factorization, assume that the input matrix is *not* permuted, and use the permutation given by *PERM* and *INVP* for factorization. The input *NAUX*, which specifies the size of double precision work array *AUX*, 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. 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 5.2.14.

7.5 WSCLDL and WSMLDL (LDL^T factorization, CSR and MSR inputs)

WSCLDL (N, IA, JA, AVALS, PERM, INVP, INFO)
void *wscldl_(int *n, int *ia, int *ja, double *avals, int *perm, int *invp, int *info)*

WSMLDL (N, IA, JA, AVALS, DIAG, PERM, INVP, INFO)
void *wsmldl_(int *n, int *ia, int *ja, double *avals, double *diag, int *perm, int *invp, int *info)*

These routines perform LDL^T factorization without pivoting, assume that the input matrix is *not* permuted, and use the permutation given by *PERM* and *INVP* for factorization. *WSPACE* of the *analyze* or *symbolic routines*, either of which must be called before a *WSxLDL* routine. Note that LDL^T factorization with Bunch-Kaufman pivoting is not available via the simple interface, and the *WSSMP* routine must be used for that.

If *NAUX* is 0, then the factor matrix is stored internally, else it is stored in *AUX*. In the latter case, the *AUX* array must be passed unaltered to any subsequent back-substitution and iterative refinement calls.

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 5.2.14.

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

WSCSVX (N, IA, JA, AVALS, PERM, INVP, B, LDB, NRHS, RCOND, INFO)
void *wscsvx_(int *n, int *ia, int *ja, double *avals, int *perm, int *invp, double *b, double *ldb, int *nrhs, double *rcond, int *info)*

WSMSVX (N, IA, JA, AVALS, DIAG, PERM, INVP, B, LDB, NRHS, RCOND, INFO)
void *wsmsvx_(int *n, int *ia, int *ja, double *avals, double *diag, int *perm, int *invp, double *b, double *ldb, int *nrhs, double *rcond, int *info)*

These routines perform LDL^T factorization without pivoting, triangular and diagonal 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. Note that LDL^T factorization with Bunch-Kaufman pivoting is not available via the simple interface, and the *WSSMP* routine must be used for that.

Upon return, *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 5.2.14.

7.7 WSSLV (solve a system using a prior factorization)

WSSLV (N, PERM, INVP, B, LDB, NRHS, NITER)
void *wsslv_(int *n, int *perm, int *invp, double *b, double *ldb, int *nrhs, int *niter)*

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

8 The Primary Message-Passing Parallel Subroutine: *PWSSMP*

The calling sequence for the parallel routine *PWSSMP* is identical to that of the serial/multithreaded 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.

8.1 Parallel modes and data distribution

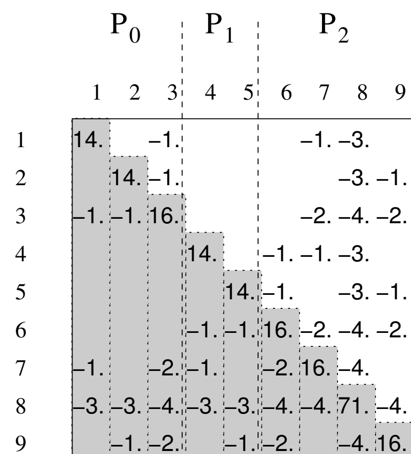
The parallel routine *PWSSMP* described in this section and the routines described later in Section 9 work in two modes; we shall call them the *0-master mode* and the *peer mode*. In the 0-master mode, all data including the entire matrix A and right-hand side B initially reside on process 0; henceforth referred to as P_0. In general, if the program is running on p MPI processes, we shall name the processes $P_0, P_1, \ldots, P_{p-1}$. In the peer mode, the initial data is distributed among the processes. 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 processes in the peer mode and the entire matrix A residing on a single process other than P_0 is a valid special case of the peer mode. In summary, 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, process 0 calls the message-passing parallel routine *PWSSMP* exactly the way it would call the serial/multithreaded routine and all the other processes call *PWSSMP* with $N_i = 0$. Therefore, the illustration of CSC-LT and MSC-LT (see Sections 5.1 and 5.2.2–5 for more details) in Figure 1 for the serial/multithreaded version is valid for the 0-master mode in the parallel version.

The example program in *pwssmp_ex1.f* at the *WSMP* home page illustrates the use of the *PWSSMP* subroutine in 0-master mode for the matrix shown in Figure 1. The example program in *pwssmp_ex3.f* at the *WSMP* home page shows the use of *PWSSMP* in the peer mode for the distribution shown in Figure 2.

As shown in Figure 2, in the default peer mode, consecutive processes 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 process P_i, then the first column on process P_{i+1} must be $l + 1$. In addition, the indices and the values corresponding to consecutive

	CSC-LT Format				MSC-LT Format				
Node#	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					



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.

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

columns must appear in consecutive order, just as in the serial/multithreaded 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 matrix can reside on any process in any order if the special routine *WSETGLOBIND* is used. This routine is described in Section 10. Unless the *WSETGLOBIND* routine and the peer mode are used the global index of a column k on process P_j is $k + \Sigma_{i=0}^{j-1} N_i$ (assuming that the columns are numbered from 1 through N).

8.2 Calling sequence

The message-passing parallel routine *PWSSMP* must be called on all the processes. The calling sequence on process 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)

void *pwssmp_(int *n_i, int ia_i[], int ja_i[], double $avals_i$[], double $diag_i$[], int perm[], int invp[], double b_i[], int *ldb_i, int *nrhs, double *aux_i, int *$naux_i$, int mrp_i[], int iparm[], double dparm[])*

In the message-passing parallel version, an argument can be either *local* or *global*. A global array or variable must have the same size and contents on all processes. The size and contents of a local variable or array vary among the processes. In the context of *PWSSMP*, global does not mean globally shared, but refers to data that is replicated on all processes. 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 5.2; this section is intended to highlight the differences between the serial/multithreaded and the message-passing versions, wherever applicable.

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

- **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. Note that if N_i = 0, then *IA*$_i$ must be a single integer with a value of 0 (with C-style numbering) or 1 (with Fortran-style numbering) to be consistent with the definition of *IA*$_i$.

- **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 process P_i. If N_i = 0, then this parameter can be a *NULL* pointer.

- **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.

- **DIAG**$_i$**:** Double precision array of size N_i (only relevant if *IPARM(4)* = 1; i.e., MSR/MSC input format is used, or if *IPARM(32)* = 1; i.e., the diagonal of the factor is requested as output). 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. Unlike most other parameters of *PWSSMP*, each process must provide space (N integers) for full *PERM* and *INVP*. After the ordering step, only process 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 processes. 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 of symbolic factorization, *PERM* and *INVP* and filled with a permutation on all processes 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 *WSSMP* routine, users of *PWSSMP* need not use the ordering produced by it, but can supply their own ordering. However, this user-supplied ordering must be the input in *PERM* and *INVP* on process P_0 for symbolic factorization and the modified ordering returned on all processes 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 8.1 *If ordering and symbolic factorization are performed in different calls, or an external non-WSMP ordering is used, then upon exiting from ordering and entering symbolic, the vectors* PERM *and* INVP *contain useful data only on process* P_0*. All processes contain the actual permutation in these vectors only after symbolic factorization and this permutation is different (but similar in properties) to the permutation on* P_0 *at the beginning of symbolic factorization.*

- **B**$_i$**, 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. The number of right-hand sides, *NRHS*, must be the same on all processes.

- **AUX$_i$ *and* NAUX$_i$:**

  AUX$_i$ and *NAUX$_i$* are obsolescent and are currently present for backward compatibility only. *NAUX$_i$* must be set to 0 on all processes.

- **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:** The description of *IPARM* and *DPARM* is contained in Sections 5.2.14 and 5.2.15, respectively. For the message-passing parallel version, all input parameters in these arrays must be identical on each process. On output, *IPARM(21), IPARM(22), DPARM(4), DPARM(5), DPARM(7),* and *DPARM(64)* are always identical on all processes. The outputs in *IPARM(24), DPARM(23),* and *DPARM(24)* are reported only on P_0. The output *IPARM(23)* is local to each process. *DPARM(2)* and *DPARM(3)*, which together report the condition number estimate in the serial/multithreaded version, always contain 0.0 on output in the message-passing 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 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 10) to define a distribution of the matrix among the processes, then you are 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.

8.3 Some parallel performance issues

Please refer to Section 3 for platform-specific performance issues and for information in setting environment variable appropriately on each platform.

The message-passing 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 process P_0. Process P_0 also bears a disproportionately high workload in the ordering phase compared to the other processes. On a heterogeneous cluster the performance of the parallel routines will be much better if process P_0 has fast cores, 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 processes 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 *IPARM(27)*. See Sections 5.2.14 and 7.1.4 for details.

Although the message-passing parallel routines can be called on any number of processes with the initial data and the final results distributed on a subset or all of them, the best performance will be obtained when the number of processes is a power of 2.

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. Since process P_0 may become a memory and communication bottleneck in the 0-master, we strongly recommend using *PWSSMP* in peer mode, especially for a large number of processes (say, 16 or more).

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 8.2 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.

9 Parallel Subroutines Providing a Simpler Interface

In this section, we list the routines that provide a simpler interface to *PWSSMP*. This interface is analogous to the simple interface to *WSSMP* described in Section 7. Parallel routines *PWSxALZ, PWSxSYM, PWSxCHF, PWSxLDL, 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/multithreaded routines *WSxALZ, WSxCHF, WSxLDL, WSxSVX,* and *WSSLV*, respectively described in Section 7. The meaning of the various parameters is the same as in the calling sequence of the *PWSSMP* routine described in Section 8. A difference between *PWSxSVX* and *WSxSVX* routines is that condition number is not estimated in the parallel routine and the output *RCOND* is always zero in *PWSxSVX*.

The example program in *pwssmp_ex2.f* at the *WSMP* home page illustrates the use of the simple parallel interface for the matrix shown in Figure 1.

Note 9.1 *The calls to the* WSSMP/PWSSMP *routines should not be mixed with those to the routines in the simple interface described here and in Section 7. The user must choose to use either the* WSSMP/PWSSMP *routines or the simple interface for a given application, and stick to the chosen interface.*

10 Miscellaneous Routines

In this section, we describe some optional routines available to the users for managing memory allocation, data distribution, and some other miscellaneous tasks. Just like other *WSMP* routines, these can be called from a C program by passing the arguments by reference (Note 5.4).

Note 10.1 *Some routines in this section have underscores in their names, and due to different mangling conventions followed by different compilers, you may get an "undefined symbol" error while using one of these routines. Placing an explicit underscore at the end of the routine name usually fixes the problem. For example, if* WS_SORTINDICES_I *does not work, then try using* WS_SORTINDICES_I_.

10.1 *WS_SORTINDICES_I (M, N, IA, JA, INFO)* ^{S,T}

This routine can be used to sort the row 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 5.2. The description of *INFO* is similar to that of *IPARM(64)*.

Please read Note 10.1 at the beginning of this section.

10.2 *WS_SORTINDICES_D (M, N, IA, JA, AVALS, INFO)* ^{S,T}

This routine is similar to *WS_SORTINDICES_I*, except that it also moves the double precision values in *AVALS* according to the sorting of indices in *JA*. The descriptions of *IA*, *JA*, and *AVALS* are similar to those in Section 5.2. The description of *INFO* is similar to that of *IPARM(64)*.

Please read Note 10.1 at the beginning of this section.

10.3 *WS_SORTINDICES_Z (M, N, IA, JA, AVALS, INFO)* ^{S,T}

This routine is similar to *WS_SORTINDICES_D*, except that the values in *AVALS* are of type *double complex*.

Please read Note 10.1 at the beginning of this section.

10.4 *WSETMAXTHRDS (NUMTHRDS)*

A call to *WSETMAXTHRDS* can be used to control the number of threads that *WSMP* spawns by means of the integer argument *NUMTHRDS*. Controlling the number of threads may be useful in many circumstances, as discussed in Section 3.4. As with all other *WSMP* functions, when calling from C, a pointer to the integer containing the value of *NUMTHRDS* must be passed. The integer value *NUMTHRDS* is interpreted by *WSMP* as follows:

If *NUMTHRDS* > 0, then *WSMP* uses exactly *NUMTHRDS* threads. If *NUMTHRDS* is 0, then *WSMP* tries to use as many cores as are available in the hardware. This is the default mode.

Note that if this routine is used, it must be called before the first call to any *WSMP* or *PWSMP* computational routine or the initialization routines (Section 10.10). Once *WSMP/PWSMP* is initialized, the number of threads cannot be changed for a given run.

The environment variable *WSMP_NUM_THREADS* can also be used to control the number of threads (Section 3.4) and has precedence over *WSETMAXTHRDS*.

10.5 *WSSYSTEMSCOPE* and *WSPROCESSSCOPE*

A call to *WSSYSTEMSCOPE* can be used to set the contention scope of threads to *PTHREAD_SCOPE_SYSTEM*. Similarly, *WSPROCESSSCOPE* can be called to set the contention scope of threads to *PTHREAD_SCOPE_PROCESS*. If these routines are used, they must be called before the first call to any *WSMP* or *PWSMP* computational routine or the initialization routines (Section 10.10). Currently, the default contention scope of the threads is *PTHREAD_SCOPE_SYSTEM*.

10.6 *WSETMAXSTACK (FSTK)*

All threads spawned by *WSMP* are, by default, assigned a 1 Mbyte stack in 32-bit mode and 4 Mbytes in 64-bit mode. In rare case, for very large matrices, this may not be enough for one or more threads. The user can increase or decrease the default stack size by calling *WSETMAXSTACK* prior to any computational or initialization routine of *WSMP*. The double precision input parameter *FSTK* determines the factor by which the default stack size of each thread is changed; e.g., if *FSTK* is 2.d0, then each thread is spawned with a 2 Mbyte stack in 32-bit mode and 8 Mbyte stack in 64-bit mode. If this routine is used, it must be called before the first call to any *WSMP* or *PWSMP* computational routine or the initialization routines (Section 10.10). In the distributed-memory parallel version, this routine, if used, must be called by all processes (it is effective on only those processes on which it is called).

Note that this routine does not affect the stack size of the main thread, which, on AIX, can be controlled by the *-bmaxstack* option during linking. Also note that when calling from a C program, a pointer to a double precision value must be passed.

On some systems, the user may need to increase the default system limits for stack size and data size to accommodate the stack requirements of the threads.

10.7 *WSETLF (DLF)*^{T,P}

The *WSETLF* routine can be used to indicate the load factor of a workstation to *WSMP* to better manage parallelism and distribution of work. The double precision input *DLF* is a value between 0.d0 and 1.d0 (0.0 and 1.0, passed by reference in C). The default value of zero (which is used if *WSETLF* is not called) indicates that the entire machine is available to *WSMP*; i.e., the load factor of the machine without the application using *WSMP* is 0. An input value of one indicates that the machine is fully loaded even without the *WSMP* application. For example, if a 2-way parallel job is already running on a 4-CPU machine, then the input *DLF* should be 0.5 and if four serial, or two 2-way parallel, or one 4-way parallel job is already running on such a machine, then the input *DLF* should be 1.0.

If this routine is used, then it must be called before the first call to any *WSMP* or *PWSMP* computational routine or the initialization routines (Section 10.10).

10.8 *WSETNOBIGMAL ()*

On most platforms, *WSMP* attempts to allocate as large a chunk of memory as possible and frees it immediately without accessing this memory. This gives *WSMP* an estimate of the amount of memory that it can dynamically allocate, and on some systems, speeds up the subsequent allocation of many small pieces of memory. However, this sometimes confuses certain tools for monitoring program resource usage into believing that an extraordinarily large amount of memory was used by *WSMP*. This large *malloc* can be switched off by calling the routine *WSETNOBIGMAL* before initializing or calling any computational routine of *WSMP* or *PWSMP*.

10.9 *WSMP_VERSION (V, R, M)*

This routine returns the version, release, and modification number of of the *WSMP* or *PWSMP* library being used in the integer variables V, R, and M, respectively.

Please read Note 10.1 at the beginning of this section.

10.10 *WSMP_INITIALIZE ()*^{S,T} and *PWSMP_INITIALIZE ()*^{P}

These routines are used to initialize *WSMP* and *PWSMP*, respectively. Their use is optional, but if used, a call to one of them must precede any computational routine. However, if any of *WSETMAXTHRDS* (Section 10.4), *WSSYSTEMSCOPE*, *WSPROCESSSCOPE* (Section 10.5), *WSETMAXSTACK* (Section 10.6), *WSETLF* (Section 10.7), and *WSETNOBIGMAL* (Section 10.8) routines are used, they must be called before *WSMP_INITIALIZE* or *PWSMP_INITIALIZE*. *PWSMP_INITIALIZE*, if used, must be called on all nodes in the message-passing parallel mode. *WSMP* and *PWSMP* perform self initialization when the first call to any user-callable routine is made.

PWSMP_INITIALIZE also performs a global communication using its current communicator, which is *MPI_COMM_WORLD* by default, unless it has been set to something else using the *WSETMPICOMM* routine. Therefore, *PWSMP_INITIALIZE* must be called on all the nodes associated with the currently active communicator in *PWSSMP*.

Please read Note 10.1 at the beginning of this section.

10.11 *WSMP_CLEAR ()*^{S,T} and *PWSMP_CLEAR ()*^{P}

Both the serial and the parallel versions of the solver have the context stored internally, which enables them to 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 example, several calls to Cholesky factorization, triangular solves, and iterative refinement can be made with different numerical 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 *WSMP_CLEAR()* in the serial/multithreaded mode and *PWSMP_CLEAR()* in the message-passing 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 7 and 9. 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. All previously stored contexts by using *WSTOREMAT* or *PWSTOREMAT* (see Section 10.17), if any, are also destroyed by a call to *WSMP_CLEAR()* or *PWSMP_CLEAR()*.

WSMP_CLEAR and *PWSMP_CLEAR* also undo the effects of *WSMP_INITIALIZE* and *PWSMP_INITIALIZE*, respectively.

Please read Note 10.1 at the beginning of this section.

10.12 *WSFFREE ()*^{S,T} and *PWSFFREE ()*^{P}

These routines are relevant only when *NAUX* = 0 and *WSSMP* or *PWSSMP* is using dynamic memory allocation for factorization.

Many applications perform ordering and symbolic factorizations only once for several iterations of factorization and solution. *WSSMP* and *PWSSMP* allocate 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 *WSFFREE ()* in serial and *PWSFFREE ()* in parallel 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.

The routines *WSFFREE ()* and *PWSFFREE ()* work only in the current context and do not affect the factor storage of other sparse systems whose contexts may have been stored using *WSTOREMAT* or *PWSTOREMAT* routines.

10.13 *WSAFREE ()*^{S,T} and *PWSAFREE ()*^{P}

These routines can be called after factorization to free the space occupied by a permuted internal copy of the coefficient matrix. Please note that if these routines are used, then iterative refinement cannot be performed because the coefficient matrix is required for computing the residual.

10.14 *WSSFREE ()*^{S,T} and *PWSSFREE ()*^{P}

The routines *WSFFREE* and *PWSFFREE* described in Sections 10.12 release the memory occupied by the factors of the coefficient matrix, but retain all other data structures to facilitate subsequent factorizations of matrices of the same size and nonzero pattern. *WSSFREE* and *PWSSFREE* release *all* the memory allocated by *WSMP* in the context of solving symmetric systems via direct factorization. If you need to solve more symmetric systems after call to *WSSFREE* or *PWSSFREE*, then you must start with the ordering or the symbolic factorization (if you are supplying your own permutation vectors) steps.

10.15 *WSSMATVEC (N, IA, JA, AVALS, X, B, IERR)*^{S}

This routine multiplies the vector X with the N-dimensional symmetric sparse matrix stored in *IA, JA, AVALS* (lower triangular part only in CSC format or upper triangular part only in CSR format) and returns the result in the vector B. The description of *N, IA, JA,* and *AVALS* is the same as in Section 5.2. *IERR* is equivalent to *IPARM(64)*, described in Section 8.2.14. Both C and Fortran style numbering convention is supported.

Unlike most other routines that work with double precision data, *WSSMATVEC* has two versions for double complex data type. *ZSSMATVEC* assumes a symmetric matrix and its interface is identical to that of *WSSMATVEC*, with the exception that *AVALS*, X, and B are double complex. An additional routine is provided for Hermitian matrices, whose interface is described below:

ZHSMATVEC (N, IA, JA, AVALS, X, B, IFMT, IERR)^{S}

ZHSMATVEC treats the triangular matrix in *IA*, *JA*, and *AVALS* as a Hermitian matrix. If the input *IFMT* is set to 0, then the triangular matrix is interpreted to be stored in CSR-UT (compressed sparse rows, upper triangular) format, and the implicit lower triangular portion is assumed to be the conjugate of the input matrix. If the input *IFMT* is set to 1, then the triangular matrix is interpreted to be stored in CSC-LT (compressed sparse columns, lower triangular) format, and the implicit upper triangular portion is assumed to be the conjugate of the input matrix.

Note that *WSSMATVEC*, *ZSSMATVEC*, and *ZHSMATVEC* routines are neither multithreaded, nor optimized for performance. Multithreaded and optimized sparse matrix-vector multiplication is included in the iterative solver package [9].

10.16 *PWSSMATVEC* (N_i, IA_i, JA_i, $AVALS_i$, X_i, B_i, *IERR*)^P

This routine multiplies the vector X with the symmetric sparse matrix stored in *IA, JA, AVALS* (lower triangular part only in CSC format or upper triangular part only in CSR format) and returns the result in the vector B. Here N_i is the local number of rows/columns on the Process i and the local number of entries of the distributed vectors X and B stored on it. The matrix as well as both the vectors are expected to be stored in a distributed fashion, similar to the distribution illustrated in Figure 2. The description of N_i, IA_i, JA_i, and $AVALS_i$ is the same as in Section 8.2. *IERR* is equivalent to *IPARM(64)*, described in Section 8.2.14. Both C and Fortran style numbering convention is supported.

Just like its serial counterpart, *PWSSMATVEC* has two versions to support double complex data type. *PZSSMATVEC* assumes a symmetric matrix and *PZHSMATVEC* assumes a Hermitian matrix. Note that *PZHSMATVEC* has the additional *IFMT* argument before *IERR* (see Section 10.15 for details).

10.17 *WSTOREMAT (ID, INFO)*^{S,T} and *PWSTOREMAT (ID, INFO)*^{P}

These routines store the current context and assign the tag *ID* to this context. *ID* is a user supplied integer input argument whose value must be in the range $0 \ldots 63$ ($0 \ldots 255$ in 64-bit mode). The purpose of these routines is to postpone further processing of the current symmetric sparse system and work on a different system. Since the input *ID* must lie in the range $0 \ldots 63$ ($0 \ldots 255$ in 64-bit mode), at most 64 (256 in 64-bit mode) systems can be stored. *INFO* is an output integer argument. *INFO* = 0 upon return signals a successful return, *INFO* = -211 indicates that the input *ID* was not in the range $0 \ldots 63$ ($0 \ldots 255$ in 64-bit mode), and *INFO* = -210 indicates that another context is already stored in the slot indicated by *ID*. After a context has been stored, it must be recalled (see Section 10.18) before another context can be stored with the same *ID*.

After a successful call to *WSTOREMAT* or *PWSTOREMAT*, the current context is destroyed; i.e., either the user should recall a previously stored context (see Section 10.18), or start a new one with the call to ordering or symbolic factorization with a valid input in *PERM*, *INVP*.

Note 10.2 *Before storing a context, it is advisable to call* WSFFREE *or* PWSFFREE *(Section 10.14) if the factors of the current matrix are not going to be used for further solves. Thus only the ordering and symbolic factorization information of the current context will be stored for future factorization/solves.*

10.18 *WRECALLMAT (ID, INFO)*^{S,T} and *PWRECALLMAT (ID, INFO)*^{P}

These routines recall the context of a sparse symmetric system that was previously stored using a call to *WSTOREMAT* or *PWSTOREMAT*. The input integer *ID* must contain the tag of the context to be recalled. A 0 integer output in *INFO* indicates a successful return. A return value of -211 in *INFO* indicates that the input *ID* was not in the range $0 \ldots 63$ ($0 \ldots 255$ in 64-bit mode). An output of -210 in *INFO* indicates that slot *ID* was empty; i.e., a context was either not previously stored with the current value of *ID* or it has already been recalled.

A successful call to *WRECALLMAT* or *PWRECALLMAT* destroys the current context and replaces it with the context previously stored with the tag *ID*. Moreover, it deallocates the memory where the context with the tag *ID* was stored. Once recalled, a context is no longer stored but becomes the current context. If processing must be postponed for the corresponding system once again, then this context must be stored again with any valid and free *ID* in the range $0 \ldots 63$ ($0 \ldots 255$ in 64-bit mode).

10.19 *WSETMPICOMM (INPCOMM)*^{P}

The message-passing parallel library *PWSMP* uses MPI_COMM_WORLD as the default communicator. The default communicator can be changed to *INPCOMM* by calling this routine.

WSETMPICOMM can be called any time and *PWSMP* will use *INPCOMM* as the communicator for all MPI calls after the call to *WSETMPICOMM*, until the default communicator is changed again by another call to *WSETMPICOMM*. Although, *WSETMPICOMM* can be called at any time, it must be used judiciously. The communicator can be changed only after you are completely done with one linear system and are moving on to another. You cannot factor a matrix

with one communicator and do the backsolves with another, unless both communicators define the same process group over the same set of nodes.

Note 10.3 INPCOMM *must be a communicator generated by MPI's Fortran interface. If you are using the* PWSMP *library from a C/C++ program and using a communicator other than* MPI_COMM_WORLD*, then you would need to use* MPI_Comm_c2f *to obtain the equivalent Fortran communicator, or write a small Fortran routine that would generate a communicator over the same processes as your C communicator.*

10.20 *WSADJSTBADPIVS (N, GAMMA)*^{S,T} and *PWSADJSTBADPIVS (N, GAMMA)*^{P}

The integer input N is the total number of equations in the system and the double precision input *GAMMA* is the value that is placed at all those diagonal locations that were smaller than *DPARM(11)* during factorization, provided that *IPARM(11)* was set to a nonzero value during factorization. Please refer to the description of *IPARM(11)* for more details.

These routines, which can be called any time after factorization, give the user another chance to control the effect of very small (and negative in case of LL^T factorization) pivots on the solution before solving for right-hand side vector(s).

10.21 *WSETGLOBIND (N*$_i$*, NUMBERING, GLI*$_i$*, INFO)*^{P}

In the parallel version, by default it is assumed that the matrix is distributed on processes P_0 to P_{p-1} in increasing order of global indices and the portion of the matrix residing on process P_i is determined by N_i on that process (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 whose calling parameters are described below. The user should note that the new global indices also determine the distribution of the right-hand side and the solution vectors.

A call to *WSETGLOBIND* affects only the current context. If used, clearly *WSETGLOBIND* must be called before any other parallel routine that involves the matrix whose distribution is determined by *GLI*$_i$.

- The *integer* input parameter N_i is the number of matrix columns/rows on process 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 9, or the transposition routines of Section 10.22 in the context of the same matrix.

- The *integer* 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.

- *GLI*$_i$ is an input *integer* array of size N_i. The k-th element of this integer array contains the global row/column number (in the overall matrix) of the k-th local row/column on process P_i. The default global index corresponding to the k-th index on P_i is $k + \Sigma_{j=0}^{i-1} N_j$. 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 processes 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 *integer* parameter *INFO* is 0 on a successful return and is -102 if the routine runs out of memory.

10.22 Routines for transposing distributed sparse matrices

In this section, we describe the routines for transposing distributed sparse matrices. While these can be used for transposing any sparse matrix, we envision them being used primarily to switch triangular portions of symmetric matrices between CSC and CSR formats.

Transposing a sparse matrix would require calls to two or three of the following routines. Note that a call to *PWS_XPOSE_IA* starts the process and is mandatory. It also clears the internal memory associated with any previously transposed matrices and allocates appropriate data structures for the new matrix. This may be followed by none or at most one call to *PWS_XPOSE_JA*. Any number of calls to *PWS_XPOSE_AV* or *PZS_XPOSE_AV* may follow.

Transposition retains the original distribution of the matrix. If there are N_i rows (columns) on process P_i in the original matrix, then the transpose will result in exactly the same N_i columns (rows) on P_i. Note that *WSETGLOBIND* (Section 10.21) can be used in conjunction with the sparse transpose routines described in this section to specify an arbitrary initial distribution and ordering of matrix row/columns. The output of the transpose will mimic the same distribution. You can use *PWSTOREMAT* (Section 10.17) and *PWRECALLMAT* (Section 10.18) to switch between matrices.

10.22.1 *PWS_XPOSE_IA (N_i, IA_i, JA_i, NNZ_i, IERR)^P*

This routine allocates some internal data structures and returns the number of nonzeros (local output NNZ_i) of the transpose that will be stored on process P_i. The global output *IERR* is the same as *IPARM(64)* described in Section 5.2.14. The description of local inputs N_i, IA_i, JA_i is the same as in Section 8.2.

PWS_XPOSE_IA must be the first routine called to transpose a matrix, the indices of whose local submatrix with N_i rows or columns on process P_i are stored in IA_i and JA_i.

A call to *PWS_XPOSE_IA* is mandatory to initiate the process of transposing a matrix.

10.22.2 *PWS_XPOSE_JA (N_i, IA_i, JA_i, TIA_i, TJA_i, IERR)^P*

This routine transposes the indices of a distributed sparse matrix. It expects local inputs N_i, IA_i, JA_i on each process P_i and generates the corresponding local output in TIA_i and TJA_i. The size of TIA_i must be at least $N_i + 1$, which is the same as the size of IA_i. The size of TJA_i must be at least NNZ_i, where NNZ_i is the number of local nonzeros in the transposed matrix, as returned by *PWS_XPOSE_IA*. The global output *IERR* is the same as *IPARM(64)* described in Section 5.2.14.

Calling *PWS_XPOSE_JA* is optional. If used, it must be called after *PWS_XPOSE_IA* without calls to any other transposition routines in between.

10.22.3 *PWS_XPOSE_AV (N_i, IA_i, JA_i, $AVALS_i$, TIA_i, TJA_i, $TAVALS_i$, IERR)^P*

PWS_XPOSE_AV, if called after *PWS_XPOSE_IA*, transposes the indices as well as the values of the distributed sparse matrix whose local portion on process P_i is stored in IA_i, JA_i, $AVALS_i$. The local output is generated in TIA_i, TJA_i, $TAVALS_i$. If it is called after the indices have already been transposed by an earlier call to *PWS_XPOSE_JA* or *PWS_XPOSE_AV*, then it transposes the values only. This routine can be called any number of times after *PWS_XPOSE_IA* or *PWS_XPOSE_JA* to transpose sparse matrices with the same structure as that of a previously transposed matrix, but different values. The size of TIA_i must be at least $N_i + 1$, which is the same as the size of IA_i. The size of TJA_i and $TAVALS_i$ must be at least NNZ_i, where NNZ_i is the number of local nonzeros in the transposed matrix, as returned by *PWS_XPOSE_IA*. The global output *IERR* is the same as *IPARM(64)* described in Section 5.2.14.

10.22.4 *PZS_XPOSE_AV (N_i, IA_i, JA_i, $AVALS_i$, TIA_i, TJA_i, $TAVALS_i$, IERR)^P*

PZS_XPOSE_AV is the complex counterpart of *PWS_XPOSE_AV*. $AVALS_i$ and $TAVALS_i$ must both be double complex in *PZS_XPOSE_AV* and double precision in *PWS_XPOSE_AV*.

10.22.5 *PWS_XPOSE_CLEAR ( )*

This routine can be used to deallocate the internal storage that is allocated in the context of transposition. Such space may be allocated by *PWS_XPOSE_IA*, *PWS_XPOSE_JA*, *PWS_XPOSE_AV*, or *PZS_XPOSE_AV*. After *PWS_XPOSE_CLEAR*, no transposition routine other than *PWS_XPOSE_IA* may be called.

11 Routines for Double Complex Data Type

The double complex (complex*16) version of the symmetric solver can be accessed via the routines *ZSSMP, ZSCALZ, ZSMALZ, ZSCSYM, ZSMSYM, ZSCCHF, ZSMCHF, ZSCLDL, ZSMLDL, ZSCSVX, ZSMSVX, ZSSLV, ZSSMATVEC* and *ZHSMATVEC* in serial/multithreaded mode and by their distributed-memory parallel counterparts *PZSSMP, PZSCALZ, PZSMALZ, PZSCSYM, PZSMSYM, PZSCCHF, PZSMCHF, PZSCLDL, PZSMLDL, PZSCSVX, PZSMSVX, PZSSLV, PZSSMATVEC* and *PZHSMATVEC* in the message-passing mode. These routines are identical to their double precision real counterparts described in Sections 5, 7, 8, and 9, with the exception that the data type of *AVALS*, *DIAG*, *B*, and *AUX* in these routines is *double complex* or *complex*16*. The *WSMP* web page at *http://www.research.ibm.com/projects/wsmp* contains example programs illustrating the use of these routines.

Note that only LDL^T factorization is supported for complex non-Hermitian matrices. While using the *ZSSMP* or *PZSSMP* routines, care must be taken to set *IPARM(31)* properly. Please refer to the description of *IPARM(31)* in Section 5.2 for more details. It must be set to 0, 1, 2, 5, or 6 for Hermitian matrices and to 3, 4, or 7 for non-Hermitian matrices. Routines *ZSCCHF, ZSMCHF, PZSCCHF*, and *PZSMCHF* can be called for Hermitian matrices only. When *ZSCLDL, ZSMLDL, ZSCSVX, ZSMSVX, PZSCLDL, PZSMLDL, PZSCSVX*, or *PZSMSVX* are called, then a symmetric non-Hermitian matrix is assumed. If you wish to perform LDL^T factorization on a Hermitian matrix, then you must use the *ZSSMP* or *PZSSMP* routines.

Upon return from the ordering or the symbolic factorization steps, the permutation vectors are *not* returned in *PERM* and *INVP* when complex matrices are being used. In fact, unlike the interface for real matrices, in the case of complex input matrices, it is not even necessary for the user to supply valid length-N integer space in *PERM* and *INVP* unless the user intends to supply their own ordering as input and skip *WSMP*'s ordering.

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

Please read the license agreement in the HTML file of the appropriate language in the *license* directory before installing and using the software. The 90-day free trial license is meant for educational, research, and benchmarking purposes by non-profit academic institutions. Commercial organizations may use the software for internal evaluation or testing with the trial license. Any commercial use of the software requires a commercial license.

13 Acknowledgements

The author would like to thank Haim Avron, Thomas George, Rogeli Grima, Mahesh Joshi, Prabhanjan Kambadur, Felix Kwok, Chen Li, and Lexing Ying for their contributions to this project.

References

[1] Haim Avron and Anshul Gupta. Managing data-movement for effective shared-memory parallelization of out-of-core sparse solvers. In *SC12 (International Conference for High Performance Computing, Networking, Storage and Analysis)*, 2012.

[2] James R. Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31:163–179, 1977.

[3] 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.

[4] 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.

[5] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, 1983.

[6] Alan George and Joseph W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, NJ, 1981.

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

[8] 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.

[9] Anshul Gupta. WSMP: Watson sparse matrix package (Part-III: Iterative solution of sparse systems). Technical Report RC 24398, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2007. *http://www.research.ibm.com/projects/wsmp*.

[10] Anshul Gupta, Mahesh Joshi, and Vipin Kumar. WSMP: A high-performance shared- and distributed-memory parallel sparse linear solver. Technical Report RC 22038, IBM T. J. Watson Research Center, Yorktown Heights, NY, April 2001. *http://www.cs.umn.edu/˜agupta/doc/wssmp-paper.pdf*.

[11] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997.

[12] Anshul Gupta, Seid Koric, and Thomas George. Sparse matrix factorization on massively parallel computers. In *SC09 (International Conference for High Performance Computing, Networking, Storage and Analysis)*, 2009.

[13] Mahesh Joshi, Anshul Gupta, George Karypis, and Vipin Kumar. Two-dimensional scalable parallel algorithms for solution of triangular systems. In *Proceedings of the 1997 International Conference on High Performance Computing (HiPC)*, 1997.

[14] Prabhanjan Kambadur, Anshul Gupta, Amol Ghoting, Haim Avron, and Andrew Lumsdaine. Modern task parallelism for modern high performance computing. In *SC09 (International Conference for High Performance Computing, Networking, Storage and Analysis)*, 2009.

[15] Joseph W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[16] Joseph W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.