RC 21886 (98462) November 16, 2000
Computer Science/Mathematics

IBM Research Report

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

Anshul Gupta

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

anshul@watson.ibm.com

IBM **Research Division**
Almaden · Austin · China · Delhi · Haifa · Tokyo · Watson · Zurich

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

Version 1.0.0β^*

Anshul Gupta

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

anshul@watson.ibm.com

IBM Research Report RC 21886 (98462)

November 16, 2000

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

Contents

1 Introduction to Part 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 on the IBM RS6000 workstations and IBM Scalable Parallel (SP) systems. It can be used as a serial package, or in a shared-memory multiprocessor environment, or as a scalable parallel solver in a message-passing environment, where each node can either be a uniprocessor or a shared-memory multiprocessor. *WSMP* is comprised of two parts, both of which are bundled in the same library. This document describes the solution of *symmetric* sparse systems of linear equations. Part I of *WSMP* replaces the older software called *WSSMP*. Part II of *WSMP* library [6], which deals with the solution of *general* sparse systems of linear equations, is described in another document that can be obtained from *ftp://ftp.cs.umn.edu/users/kumar/anshul/wsmp2.ps* via anonymous FTP or through your web browser. Technical papers related to the software and some example programs (also in Appendices A–E) can be obtained from the site *ftp://ftp.cs.umn.edu/users/kumar/anshul*.

For solving symmetric systems, *WSMP* uses a modified version of the multifrontal algorithm [11] for sparse Cholesky factorization and a highly scalable parallel sparse Cholesky factorization algorithm [9, 3]. The package also uses scalable parallel sparse triangular solvers [10] and an improved and parallelized version of the previously released package *WGPP* [4, 5] for computing fill-reducing orderings. Sparse symmetric factorization in *WSMP* has been clocked at up to 3.6 GFLOPS on an RS6000 workstation with four 375 MHz Power3 CPUs and 90 GFLOPS on a 128-node SP with two-way SMP 200 MHz Power3 nodes. For more details on the implementation and performance of *WSMP* for solving symmetric sparse systems, please refer to [8].

In this document, the term *node* refers to a uniprocessor or a multiprocessor computing unit with shared memory. A node may consist of one or more *processors* or CPUs. When *WSMP* runs in a parallel environment on an SP computer, nodes communicate with other nodes via message-passing using Message Passing Interface (MPI) [13] over the SP high-speed switch. The parallelism within multiprocessor nodes is exploited through the POSIX threads library [12]. In fact, it is possible to run multiple processes on a single node. The processes communicated with each other via MPI and each process may itself be parallel and may spawn multiple threads if running on a multiprocessor node. The use the multithreaded solver is completely transparent to the user. The serial and the multithreaded codes are packaged together. The same routines work in serial mode on a platform with a single CPU and in multithreaded mode on a symmetric multiprocessor (SMP) platform.

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*. Neither the serial nor the parallel version has out-of-core capabilities and the problems must fit in the main memory for reasonable performance. The factorization routines follow a predetermined pivot sequence that orders the sparse matrix for fill-reduction purpose only.

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}.

The organization of this document is as follows. Section 2 lists the various libraries that are available and describes how to obtain and link the libraries. Section 3 describes the changes in the latest version that may affect the users of earlier versions. Section 5 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 6 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. In this section, the differences from the message-passing parallel version are noted, wherever applicable. 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 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 6 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 to provide some extra flexibility in memory management and data distribution. Section 11 contains the terms and conditions that all users of the package must adhere to.

2 Obtaining, Linking, and Running with *WSMP*

The *WSMP* software is packaged into two libraries. The serial and multithreaded single-node routines are a part of the *WSMP* library. This library can be used on single-CPU or SMP workstations. The second library is called *PWSMP*, which is required in addition to the *WSMP* library on a distributed-memory platform such as an SP or a network of RS6000 workstations. In addition to Sections 2.3 and 2.5, please refer to Appendices A–E for examples on using the libraries.

2.1 Compiler and OS requirements

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

1. XLF Fortran compiler version 6.1.0.3 or higher.
2. XLC C language compiler version 4.4.0.0 or higher.
3. AIX version 4.3.2 or higher.

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

2.2 Obtaining the software

The package is available in the form of two libraries. The *WSMP* libraries (*libwsmp.a* or *libwsmpp3.a*) are meant to be used on a serial or a shared-memory parallel workstation. On a message-passing distributed machine, such as the SP, the *PWSMP* library (*libpwsmp.a* or *libpwsmpp3.a*) must be linked in addition to the *WSMP* library.

The libraries *libwsmp.a* and *libpwsmp.a* work on all RS6000 workstations and SP parallel computers, whereas *libwsmpp3.a* and *libpwsmpp3.a* are compiled specifically for Power3 microprocessors and are usually faster on the Power3 architecture. Users of Power2 hardware should continue to use the *WSSMP/PWSSMP* libraries *libwssmpp2.a* and *libpwssmpp2.a* available at *ftp://ftp.cs.umn.edu/users/kumar/anshul/wssmp-46.tar* because *WSMP* does not have a version optimized for Power2.

The libraries can be obtained by sending e-mail to *anshul@watson.ibm.com* with the subject line containing the library name followed by one or more blanks followed by the e-mail address at which the appropriate library and the accompanying documentation will be e-mailed. The body of the message must contain the requester's name, affiliation, and the intended use of the software. The users must read Section 11 of this document before using the software. The documentation is sent in the postscript format and the libraries are e-mailed as uuencoded files, which should yield binary files with *.a* extensions upon successful decoding. You may need to run *ranlib* on the decoded files. If you are using older versions of XLF or AIX than the ones for which the libraries are available, please send e-mail to *anshul@watson.ibm.com* with your AIX and XLF versions and we will attempt to make a library suitable for your system.

2.3 Linking the libraries

The *WSMP* and the *PWSMP* libraries contain code written in both C and FORTRAN. The libraries can be linked with either C or FORTRAN programs using either *xlf* or *xlc* commands (or their variations). On an SP with uniprocessor nodes, *mpcc* or *mpxlf* commands should be used for compiling and linking so that compiler can find the paths to the appropriate include files, such as *mpi.h, mpif.h*, and the appropriate MPI library is linked. On an SP with SMP nodes, the reentrant versions of these commands; i.e., *mpcc_r* or *mpxlf_r* should be used.

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

1. A Power3 workstation:

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

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

2. An SP parallel computer with Power3 nodes:

   mpxlf_r <source/object files> -o <executable target> -lm_r -lpthreads -lwsmpp3 -lpwsmpp3 -L<path to parent directory of *libwsmpp3.a* and *libpwsmpp3.a*>

   mpcc_r <source/object files> -o <executable target> -lxlf90_r -lm_r -lpthreads -lwsmpp3 -lpwsmpp3 -L<path to parent directory of *libwsmpp3.a* and *libpwsmpp3.a*>

Note 2.2 *On most machines, you will need to use the* -bmaxdata:<*membytes*> *option while linking so that the data segment of the program can use all the available memory. Here* membytes *is the integer number of bytes of memory that you want your program to be able to allocate. If you get an error code which, modulo 1000, is -102 (see description of* IPARM(64) *in Section 6.2.14 for more details) when you are not expecting the program to run out of memory, then using this option or increasing* membytes *may solve the problem.*

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

Note 2.4 *Although* WSMP *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.*

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

2.4 Using WSMP in conjunction with other IBM software

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

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

2. *WSMP* performs self initialization when the first call to any user-callable *WSMP* routine is made. However, due to some common non-user-callable routines between *WSMP* and other IBM libraries, you may get a run time error if such a routine is called before *WSMP* has been initialized. A call to *WSMP_INITIALIZE()* or *PWSMP_INITIALIZE()* (Section 10.1) at the beginning of your program (before any library call) eliminates the possibility such interference.

2.5 Setting environment variables and running

In a shared memory environment, the user may want to experiment with setting different values for YIELDLOOPTIME and SPINLOOPTIME to see which values yield the best performance on their system. Generally recommended values are YIELDLOOPTIME=500 and SPINLOOPTIME=500, but optimal values may vary depending on the type of machine and the number of CPUs. On an SP with shared-memory nodes, the YIELDLOOPTIME and SPINLOOPTIME environment variables must be set to the same value on each node.

The following environment variables must be set to get the best performance on a message-passing machine. The most important of these are MP_EUILIB = us, which selects the user-space mode and MP_EUIDEVICE = css0, which selects the high-performance switch for inter-node communication.

ksh	**csh** or **tcsh**
export MP_EUILIB=us	setenv MP_EUILIB us
export MP_EUIDEVICE=css0	setenv MP_EUIDEVICE css0
export MP_CSS_INTERRUPT=no	setenv MP_CSS_INTERRUPT no
export MP_PULSE=0	setenv MP_PULSE 0

Inter-node communications can be performed over the LAN by changing MP_EUILIB to *ip* and MP_EUIDEVICE to *en0*, thereby enabling a network of RS6000 workstations with the POE (Parallel Operating Environment) installed to operate as a parallel computer. Note that MP_EUILIB = *us* cannot be used with MP_EUIDEVICE = *en0*; however, MP_EUILIB may be set to *ip* with MP_EUIDEVICE = *css0*.

Note that all *WSMP* and *PWSMP* libraries use Pthreads to exploit possible SMP parallelism on each node. It is best to use only one MPI process per node as long as each node has up to 4 or 8 CPUs. Usually, for 8 or more CPU's per node, it usually pays off to run more than 1 MPI processes per node; e.g., on a 4-node SP with Nighthawk-2 (16-way SMP Power3) nodes, the best performance may be obtained by running 8 MPI processes—two on each node. In this situation, by default, each process will attempt to use all 16 CPUs on a node. However, the user must limit the number of CPU's that each process uses to 8. This can be accomplished by using the *WSETMAXTHRDS* routine. Please refer to Section 10.9 for more details on selecting the number CPU's to be used by each process. Also, while running multiple MPI processes per node, setting the environment variable MP_SHARED_MEMORY to "yes" improves message-passing performance between processes running on the same node.

Performance may not be the only reason for chosing to run more than one process per node. Many Power3 nodes have more than 2 Gbytes of RAM, which a 32-bit software such as *WSMP* cannot use in a single process. Running multiple processes per node allows one to use more than 2 Gbytes of memory per node.

For best performance, the processes should be run in user-space mode (i.e., the environment variable MP_EUIDEVICE must be set to *us*) and no more than 4 processes should be run on one node.

2.6 Linking and execution-time problems

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

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

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

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

3 Changes from Earlier Versions

The user interface of version 1.0 of *WSMP* is mostly identical to version 4.6 of *WSSMP* [7]. There is one major enhancement, which is the addition of *WSTOREMAT*, *PWSTOREMAT*, *WRECALLMAT*, and *PWRECALLMAT* routines. These allow users to work with more than one sparse symmetric linear systems at a time. Please refer to Section 10 for more details.

4 OSL Support

If you are a user of Version 2 of OSL, then you can use *WSMP* and *PWSMP* libraries to automatically parallelize the interior-point code of OSL on both shared-memory and message-passing passing AIX platforms. The following sections explain the actions that you need to take on various AIX platforms if you want to use parallelization to accelerate OSL's interior-point performance.

4.1 A Serial AIX Workstation

Do nothing. The serial version of *WSSMP* is already included in Version 2 of OSL.

4.2 An SMP AIX Workstation

Just link your program with the *WSMP* library and place "-lwsmp" *before* "-losl" while creating your executable. Also, make sure that you use the reentrant version of the compiler for linking (e.g. xlC_r, xlc_r, xlf_r, etc.). The following is an example of correctly creating the executable *foo* from your source file *foo.c*:

xlc_r -o foo foo.c -lwsmp -losl -L<path to the directory where the libraries reside> -lxlf90_r -lm_r -lpthreads

In the above example, using "xlc" instead of "xlc_r" or using "-losl -lwsmp" instead of "-lwsmp -losl" will lead to run time problems.

4.3 An SP Parallel Computer

Although Version 2 of OSL is meant to run on a serial workstation, extensions embedded in the *PWSMP* library enable its users to take advantage of the IBM SP parallel computers for solving large linear programming problems. Moreover, if your SP has multiprocessor nodes, then *WSMP* and *PWSMP* allow the interior-point solvers in OSL to use both levels of parallelism. Since the parallel version runs in 0-master mode (Section 8.1), don't expect spectacular scalability on dozens of nodes. But it should still yield reasonable speedups on large problems.

In order to run OSL Version 2 on an SP, you need to link your program with both *PWSMP* and *WSMP* libraries in the sequence "-lpwsmp -lwsmp -losl." Additionally, you need to place a call to the function *parosl_initialize()* as the first executable statement of your program and *parosl_terminate()* as the last executable statement of your program. Figure 1 shows simple C and Fortran examples. Also you need to use "mpxlf" or "mpxlc" to compile and link your program ("mpxlf_r" or "mpxlc_r" if the SP has multiprocessor nodes). The following is an example of correctly creating the executable *foo* from your source file *foo.c* after you have added calls to *parosl_initialize()* and *parosl_terminate()* at the beginning and end of your main program.

mpxlc_r -o foo foo.c -lpwsmp -lwsmp -losl -L<path to the directory where the libraries reside> -xlf90_r -lm_r

Now you can run your program on multiple nodes of an SP using the "poe" or "llsubmit" commands.

5 Overview of Functionality

WSSMP and *PWSSMP* are the primary routines in *WSMP* for solving symmetric sparse systems of linear equations. Both the serial and the 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*

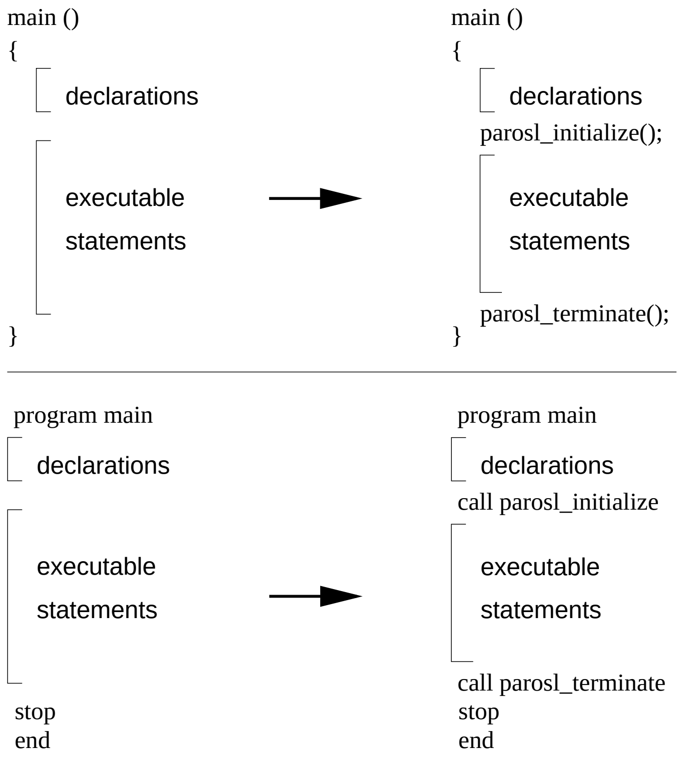


Figure 1: Modifying C and Fortran programs to use OSL on an SP.

and *PWSSMP* (described in Sections 6 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 6 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.

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.

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.

5.1 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. The original matrix is not altered; the permutation is stored in the vectors *PERM* and *INVP*. Please refer to Sections 6.2.6 and 6.2.7 for a detailed description of these vectors.

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.

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 node 0 only.

5.2 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 nodes. 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.

5.3 Numerical factorization

The numerical factorization performs either LL^T or LDL^T factorization on the input matrix. 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 6.2.14.

Once a symbolic factorization is step has been performed, numerical factorization can be called any number of times for matrices with identical nonzero pattern (determined by *IA* and *JA*) but possibly different numerical values in *AVALS*.

The factor matrix that is produced as a result of Cholesky or LDL^T factorization is either stored internally or in the *AUX* array, if such an array of sufficient capacity is provided. In the latter case, this array must be passed unaltered to all subsequent calls to back substitution and iterative refinement routines.

5.4 Back substitution

The back substitution or the triangular solve phase generates the actual solution to the system of linear equations. This phase requires the factor generated by a previous call to numerical factorization. The factor may either be stored internally or passed in the *AUX* array, depending on which option was used during 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.

5.5 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 most problems, this step is not necessary.

6 The Primary Serial Subroutine: *WSSMP*

This section describes the use of the *WSSMP* subroutine and its calling sequences in detail. There are five basic tasks that *WSSMP* is capable of performing, namely, *ordering*, *symbolic factorization*, 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 6.2). 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 6.2.14 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.

6.1 Types of matrices accepted and their input format

The *WSSMP* routine works for symmetric positive-definite, quasi-definite, and indefinite matrices whose factorization is numerically stable irrespective of the pivot sequence. For certain kinds of indefinite systems, special ordering techniques must be used (see Section 7.1 for more details). It can also handle semi-definite matrices via appropriate use of the options provided by *IPARM(11:13)*, and *DPARM(11,12,21,22)*. Please refer to Section 6 for details. Permutation of rows and columns is performed for fill-reduction only; the factorization routines do not perform pivoting based on numerical criteria. 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 and all integers must be 4 bytes long.

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 2 illustrates both input formats; they are also explained briefly in Sections 6.2.2, 6.2.3, 6.2.4, and 6.2.5.

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 6.2.14 for more details.

6.2 Calling sequence of the WSSMP subroutine

There are five types of arguments, namely input (type $\mathbf{I}$), output (type $\mathbf{O}$), modifiable (type $\mathbf{M}$), temporary (type $\mathbf{T}$), and reserved (type $\mathbf{R}$). 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 6.1 *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.

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

A 9 X 9 symmetric sparse matrix.

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

	CSC-LT Format			MSC-LT Format			
K	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				

Figure 2: Illustration of the two input formats for the serial and multithreaded *WSSMP* routines.

Note 6.2 *The original code for* WSSMP *is in FORTRAN and expects the parameters to be passed by reference. Therefore, when calling* WSSMP *from a C program, the addresses of the parameters described in Section 6.2 must be passed.*

The calling sequence and description of the parameters of *WSSMP* is 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 in Appendix A illustrates the use of the *WSSMP* subroutine for the matrix shown in Figure 2.

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

6.2.1 N (type I): matrix dimension

```
INTEGER N
```

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

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

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

IA is an integer array of size one greater than N. *IA(I)* points to the first column index of row I in the array *JA*. Refer to Figure 2 and description of *IPARM(4)* in Section 6.2.14 for more details.

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

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

```
INTEGER 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.

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

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

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

The array *AVALS* contains the actual double precision values corresponding to the indices in *JA*. The size of *AVALS* is the same as that of *JA*. See Figure 2 for more details. The input *AVALS* may be modified if scaling is opted for; see the description of *IPARM(10)* for more details.

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

AVALS is not accessed during the ordering and symbolic phases.

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

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

If the MSR input format is used, then *DIAG(I)* contains the I-th diagonal element of the coefficient matrix. If CSR/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). The input *DIAG* is not modified even if scaling is opted for because scaling does not modify the user input values if MSR format is used; instead it modifies an internal copy. See the description of *IPARM(10)* for more details.

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

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.

DIAG is accessed only in the factorization phase if either *IPARM(4)* is 1 or *IPARM(32)* is 1.

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

```
INTEGER PERM ( N )
```

PERM is the permutation vector, as defined in Sparspak [1]. If *J* = *PERM(I)*, then the J-th row and column of the original matrix become the I-th row and column in the permuted matrix to be factored. If ordering is one of the steps that the call to *WSSMP* is made to perform, the *PERM* is an output parameter. If *WSSMP* is called to perform a task or a set of tasks other than ordering, then *PERM* is an input parameter if either the matrix A of the right-hand side B is not permuted already. If both A and B are already permuted when passed in to *WSSMP*, then *PERM* is not referenced. See Section 6.2.14 for more details.

Note 6.3 *If ordering and symbolic factorization is 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.*

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

```
INTEGER INVP ( N )
```

INVP is the inverse permutation vector, as defined in Sparspak [1]. If *J* = *INVP(I)*, then the I-th row and column of the original matrix become the J-th row and column in the permuted matrix to be factored. If ordering is one of the steps that the call to *WSSMP* is made to perform, the *INVP* is an output parameter. If *WSSMP* is called to perform a task or a set of tasks other than ordering, then *INVP* is an input parameter if either the matrix A of the right-hand side B is not permuted already. If both A and B are already permuted when passed in to *WSSMP*, then *INVP* is not referenced. See Section 6.2.14 for more details. Also refer to Note 6.3.

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

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

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

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

```
INTEGER LDB
```

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

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

```
INTEGER NRHS
```

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

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

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

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

Note 6.4 *If* AUX *is used in a call to* WSSMP *for Cholesky factorization, then it must be passed unaltered in subsequent calls to* WSSMP *for triangular solves and iterative refinement with the same matrix.*

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

```
INTEGER NAUX
```

NAUX is the number of double words in *AUX*. If *NAUX* is zero, then *WSSMP* allocates space dynamically for factorization. If *NAUX* is a positive integer, then *WSSMP* attempts to use the first *NAUX* double words of the array *AUX* for factorization. If that is not sufficient, then *WSSMP* returns without completion. *IPARM(64)* returns the status of the call and tells the user whether the *WSSMP* call was successful or not. Whether *NAUX* is 0, or insufficient, or more than sufficient, *IPARM(23)* returns the actual size of *AUX* required. *IPARM(23)* is filled with the correct value of *NAUX* after symbolic factorization and the user has a chance to inspect this value before calling *WSSMP* for Cholesky factorization. See Section 6.2.14 for more details.

6.2.13 MRP, (type O): pivot info

```
INTEGER MRP ( N )
```

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(11)* during factorization. Otherwise, *MRP(I)* is not touched. Please refer to the description of *IPARM(11)* for more details.

MRP is accessed during factorization only if *IPARM(11)* $= 2$.

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

```
INTEGER IPARM ( 64 )
```

IPARM is an integer array of size 64 that is used to pass various optional parameters to *WSSMP* and to return some useful information about the execution of 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

	IPARM			DPARM		
Index	Default	Description	Type	Default	Description	Type
1	mandatory I/P	default/user defined	M	-	unused	-
2	mandatory I/P	starting task	M	-	matrix norm^S	O
3	mandatory I/P	last task	I	-	mat. inv. norm^S	O
4	0	I/P format	I	-	max. fact. diag.	O
5	1	numbering style	I	-	min. fact. diag.	O
6	1	no. of iter. ref.	M	10^{-10}	rel. err. lim.	I
7	0	residual norm type	I	-	relative err.	O
8	0	mat. permut. option	M	-	unused	-
9	0	RHS permut. option	M	-	unused	-
$10^{S,T}$	0	scaling option	M	10^{-2}	scaling range	I
11	0	bad pivot handling	I	4×10^{-16}	bad piv. threshold	I
12	0	small piv. handling	I	10^{-14}	small piv. thresh.	M
13	0	bad pivot flag	I	-	unused	-
14	0	AVALS reuse opt.	M	-	unused	-
15	0	N1	I	-	unused	-
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	-	small pivot count	O	10^{-7}	small piv. subst.	I
23	-	minimum NAUX	O	-	factorization ops.	O
24	-	factor nonzeros	O	-	tri. solve 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^P	0	enable dynamic dist.	I	-	unused	-
30	0	tri. solve opt.	I	-	unused	-
31	0	LL^T or LDL^T factor	I	0.0	fact. opt. thresh.	I
32	0	diag. O/P opt.	I	-	unused	-
33	-	no. of CPU's used	O	-	unused	-
34-63	-	reserved	R	-	reserved	R
64	-	info O/P	O	-	first bad pivot	O

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

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.

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

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

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

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

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

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

  IPARM(3) must contain the number of the last task to be performed by *WSSMP*. In a call to *WSSMP*, all tasks from *IPARM(2)* to *IPARM(3)* are performed (both inclusive). If *IPARM(2)* > *IPARM(3)* or both *IPARM(2)* and *IPARM(3)* is out of the range 1–5, then no task is performed. This can be used to fill *IPARM* and *DPARM* with default values; e.g., by calling *WSSMP* with *IPARM(1)* = 0, *IPARM(2)* = 0, and *IPARM(3)* = 0.

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

  IPARM(4) denotes the format in which the coefficient matrix A is stored. *IPARM(4)* = 0 denotes CSR/CSC format and *IPARM(4)* = 1 denotes MSR/MSC format. Both formats are illustrated in Figure 2.

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

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

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

  On input 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.

  Note 6.5 *In the message-passing parallel version, the relative error 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 error computation, then these must be performed in batches of at most 20 each.*

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

  If *IPARM(7)* $= 0$, 1, 2, or 3, then the residual in iterative refinement is computed in double precision (the same as the remainder of the computation). If *IPARM(7)* $= 4$, 5, 6, or 7, then the residual in iterative refinement is computed in quadruple precision (which is twice the precision of the remainder of the computation). If *IPARM(7)* $= 0$ or 4, then exactly *IPARM(6)* number of iterative refinement steps are performed without checking for relative error. If *IPARM(7)* $= 1$, 2, 3, 5, 6, or 7, then iterative refinement is performed until the number of iterative refinement steps is equal to *IPARM(6)* or until the relative error given by $||b - A\overline{x}||/||b||$ falls below the input value in *DPARM(6)*. Here A is the coefficient matrix, $\overline{x}$ is the computed solution, and b is the right-hand side. If *IPARM(7)* $= 1$ or 5, then 1-norms are used in computing the relative error, if *IPARM(7)* $= 2$ or 6, then 2-norms are used, and if *IPARM(7)* $= 3$ or 7, then infinity-norms are used. Moreover, if *IPARM(7)* $= 1$, 2, 3, 5, 6, or 7, then the actual relative error given by $||b - A\overline{x}||/||b||$ at the end of the last iterative refinement step is placed in *DPARM(7)*. In *WSSMP*, even if iterative refinement is not performed (i.e. if *IPARM(6)* $= 0$ or if *IPARM(3)* < 5) the value of relative error is computed and placed in *DPARM(7)* if *IPARM(7)* $= 1$, 2, 3, 5, 6, or 7. However, in *PWSSMP*, relative error is computed only if the iterative refinement step is performed. Relative error 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 errors amongst the *NRHS* solution vectors is considered. Also note that, if scaling is opted for (i.e., if *IPARM(10)* $= 1$), then the relative errors are computed with respect to the scaled system and not the original system.

  In the message-passing parallel version, the relative error is not computed and iterative refinement is not performed if *NRHS* > 20. Please refer to Note 6.5.

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

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

  IPARM(8) $= 1$ means that the matrix is already permuted by the permutation vectors in *PERM* and *INVP* and it must be used as is by *WSSMP*.

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

  Note that if the input format is MSR/MSC (i.e. *IPARM(4)* is 1) and *DIAG* is being used as an output (i.e. *IPARM(32)* is 1), then *IPARM(8)* $= 2$ cannot be used. Also, *IPARM(8)* $= 2$ cannot be used in the peer mode of the message-passing parallel version because the sizes of the original and the permuted *JA*, in general, do not match.

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

  On input, *IPARM(9)* $= 0$ means that the right-hand side B is not permuted and its permutation given by *PERM* and *INVP* must be used by *WSSMP*. *IPARM(9)* $= 1$ means that B is permuted already and must be used as it is by *WSSMP*. *IPARM(9)* $= 2$ means that the B not permuted and instructs *WSSMP* to overwrite

B with its permuted version. In addition, if *IPARM(9)* $=$ 2 on input, it is set to 1 on output. If *IPARM(9)* $=$ 1 on input and ordering is one of the steps being performed, then *IPARM(9)* is treated as if it was equal to 2 on input. If a solution is obtained or iterative refinement is performed, then the solution is in the original unpermuted order if *IPARM(9)* $=$ 0 and it is permuted if *IPARM(9)* is 1 or 2 on input. The input *RHS* is overwritten by the permuted *RHS* only if the triangular solutions are performed.

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

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

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

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

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

  This option is currently unavailable in the message-passing parallel version.

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

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

  The slight difference in the handling of bad pivots in case of LDL^T factorization is as follows. In this case, the absolute value of the pivot is compared with *DPARM(11)* and corrective action taken as desired. If *IPARM(11)* is 1 or 2 and the absolute value of the pivot is less than or equal to *DPARM(11)*, then it replaced by *DPARM(21)* with the sign of the original pivot. Note that both *DPARM(11)* and *DPARM(21)* must always be non-negative.

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

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

One again, in the case of LDL^T factorization, the absolute value of the pivot is compared against *DPARM(11)* 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(11)* and *DPARM(21)*, *DPARM(12)* and *DPARM(22)* must always be non-negative.

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

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

- **IPARM(14), type 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.

  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* 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. This *is not* recommended in message-passing parallel version as the original values are discarded anyway at a slightly later stage so copying them in *AVALS* wastes time.

  If *IPARM(14)* is 2, then *WSSMP* uses the space in *JA* to store a permuted internal copy of the matrix indices. If the input format is MSR (i.e., *IPARM(4) = 1*), then this option should be used only if *JA* has the space for at least N extra integers. This *is* recommended in message-passing parallel version in the 0-master mode (see Section 8.1 for description of distributed modes) as it saves space on process 0 when the entire original matrix resides on process 0.

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

  If *IPARM(14)* is 4, then *WSSMP* decides between options 0 and 1 for this parameter depending on the memory requirements and availability of the given problem and machine combination. Upon return from a factorization call, *IPARM(14)* is set to either 0 or 1 depending on the option chosen by *WSSMP*.

  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 6.6 *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), type I:**

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

  This option 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.

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

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

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

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

  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), type I:**

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

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

  The input *IPARM(20)* lets the user communicate some known characteristics of the sparse matrix to *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 following categories, then the default value 0 should be used. Certain sparse matrices have a a very irregular structure and 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 *IPARM(20)* should be 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.

Note 6.7 Recommended options for the serial 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 or* IPARM(17) *should contain* $N + 1$*, where* N *is the dimension of the system. The ordering speed can be further increased by using a higher value (2 or 3) in* IPARM(16).

Note 6.8 Recommended options for the parallel version: *The recommended contents of* IPARM(16:20) *in the parallel version are (1,0,0,0,1) for linear-programming problems, (1,0,0,0,2) for finite-element matrices that can benefit from graph compression, and (1,0,0,0,0) otherwise. The ordering process can be speedup up by making* IPARM(17) $= K$*, where* $K = N/p - 1$*,* N *is the dimension of the system and* p *is the number of processes being used. The ordering speed can be further increased by using a higher value (2 or 3) in* IPARM(16).

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

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

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

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

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

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

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

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

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

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

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

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

- **IPARM(26)**^{P}**, 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)^{P,T}, 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)* $= 0$ allows the fill-reducing ordering to determine load-balancing among the processes with only a slight adjustment for load balancing. 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. Our experience has shown that using *IPARM(27)* $= 1$ is significantly beneficial for factoring LP (Linear Programming) matrices. The default value of *IPARM(27)* is 0.

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

  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)^{P}, 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 an load of different nodes in a heterogeneous machines. 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(30), 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 (i.e., *IPARM(31) = 0*), calling *WSSMP* or *PWSSMP* with *IPARM(2)* $= 4$, *IPARM(3)* $= 4$, and *IPARM(30)* $= 0$ should produce the same output in B as a call with *IPARM(30)* $= 1$ followed by a call with *IPARM(30)* $= 2$. If forward and back solves are performed separately, then the user looses the ability to perform iterative refinement on the solution thus obtained.

  If LDL^T factorization is performed, then 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), type I:**

  This input is read during the numerical factorization phase. If *IPARM(31)* is 0, then LL^T factorization is performed and if *IPARM(31)* is 1, then LDL^T factorization is performed. If *IPARM(31)* is 2, then each diagonal entry of the coefficient matrix A is compared against *DPARM(31)* and if any of the diagonal entries is less than or equal to *DPARM(31)*, then LDL^T factorization is performed, else LL^T factorization is performed.

  During condition number estimation, solution, and iterative refinement steps the right algorithm is automatically automatically selected based on the type of factorization performed and no additional intervention on part of the user is required.

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.

- **IPARM(32), type I:**

  WSSMP does not give user the access to the factors of the coefficient matrix, which is stored in a non-standard format either in the user supplied storage area *AUX*, or in internally allocated data structures. However, occasionally the user might want to inspect the diagonal values of the factor to analyze the numerical accuracy of the factorization process. If *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, then the output actually contains the inverse of the diagonal D.

- **IPARM(33), type O:**

  On output, *IPARM(33)* is set to the number of CPU's that were used on the node/workstation in SMP mode. This value is the minimum of the the number of CPU's physically present on the node/workstation and the number provided by the user in a call to the *WSETMAXTHRDS* routine (Section 10.9), if such a call is made before calling *WSSMP/PWSSMP*. If *WSETMAXTHRDS* is not called, then all the CPU's on a node/workstation are used. If *IPARM(33)* returns a number smaller than the number of CPU's on a node/workstation and the user has not made a call to *WSETMAXTHRDS*, then a hardware malfunction could be preventing the code from using all the CPU's.

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

- **IPARM(34:63), type R:**

  These are reserved for future use.

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

  In the event of a successful return from *WSSMP*, *IPARM(64)* is set to 0 on output. A nonzero value of *IPARM(64)* upon output indicates that *WSSMP* did not complete execution and detected an error condition. There are two types of error codes—negative and positive. In *PWSSMP*, the error code returned on all nodes is identical. In case of an error with a negative error code, the smallest node number on which the error was detected is multiplied by 1000 and is subtracted from the error code. For example, an error code of -102 indicates that node 0 detected error -102 (though *IPARM(64)* is set to -102 on all nodes upon return), and an error code of -2700 indicates that node 2 detected error -700.

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

  If the user supplies the memory for factorization in *AUX* and the size *NAUX* of this array is not sufficient, then *IPARM(64)* is -101 on output.

  If dynamic memory allocation by *WSSMP* fails (which can happen even if the user supplies factorization space in *AUX* because *WSSMP* allocates more memory for internal book-keeping), then *IPARM(64)* is set to -102 on return. This is one of the most common error codes encountered by the users and can often be corrected by the appropriate use of the *-bmaxdata* option while linking (see Note 2.2 for more details).

  An output value of -200 for *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. An output value of -700 for *IPARM(64)* indicates an internal error and should be reported to *anshul@watson.ibm.com*.

  Positive Error Codes: A positive integer value of *IPARM(64)* between 1 and N on output indicates a computational error. In this case, *IPARM(64)* is the index of the first pivot that was less than or equal to

DPARM(11) before computing the square root for Cholesky factorization (if LDL^T factorization is being performed, then it is the index of the first pivot whose absolute value is less than or equal to *DPARM(11)*). If C-style (0-based) indexing is used and *IPARM(64)* > 0, then *IPARM(64)* is 1 + the index of the bad pivot.

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

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

The entries *DPARM(33)* through *DPARM(63)* are reserved. Unlike *IPARM*, only a few of the first 32 entries of *DPARM* are used. The description of only the relevant entries of *DPARM* is given below.

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

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

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

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

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

  On output, this contains the largest diagonal element encountered in the factorization process (just before the square root in LL^T factorization). 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), type O:**

  On output, this contains the smallest diagonal element encountered in the factorization process (just before the square root in LL^T factorization). The maximum is taken before the diagonal value is tested for conditions described in the description of *IPARM(10,11)* and if the factorization proceeds normally without a diagonal entry 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), 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 error becomes less than *DPARM(6)*. *DPARM(6)* is not used if *IPARM(7)* $=$ 0.

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

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

  In the message-passing parallel version, the relative error 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 error computation, then these must be performed in batches of at most 20 each.

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

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

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

  This is the lower threshold on value of a good diagonal value. If a pivot value is less than or equal to *DPARM(11)*, then either an error is flagged or corrective action is taken as specified by *IPARM(11)*. *DPARM(11)* must be non-negative. In case of LDL^T factorization, *DPARM(11)* 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.

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

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

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

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

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

  Please refer to the description of *IPARM(12)*. If *DPARM(21)* is being used; i.e., if *IPARM(12)* = 1, then it is the user's responsibility to initialize *DPARM(21)* with a valid positive double precision value. *DPARM(22)* must be non-negative.

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

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

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

  This contains the number of floating point operations required in forward and backward elimination for a single right-hand side vector. The total operation count for triangular solution is the product of *NRHS* and *DPARM(24)*.

- **DPARM(31), type O:**

  Please refer to the description of *IPARM(31)*.

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

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

7 Subroutines Providing a Simpler 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 6, *unless mentioned otherwise*. Appendix B contains an example driver program that uses the simple interface. Appendix F gives the source code of many of the simple subroutines to show how they are implemented via calls to *WSSMP*. 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.*

7.1 WMMRB, WKKTORD, WN1ORD (ordering)

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

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

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

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

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

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

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)
```

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)
```

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)
```

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 Note 2.2), 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 7.2 *Note that* WSSMP*, by default, passes (1,0,1,0,0) in* OPTIONS *to* WMMRB *(Table 1, Section 6.2). 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 7.3 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), type I:**

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

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

  OPTIONS(2) specifies the maximum number of nodes that a subgraph must have before it is ordered by using a minimum local fill algorithm without further subpartition. The user can obtain a pure minimum local fill ordering by specifying *OPTIONS(2)* greater than *N*. A value of 0 in this field lets *WMMRB* chose its own default. Typically, it is best to use the default, but advanced users may experiment with this parameter to find out what best suits their application. Sometimes a value larger than the default, which is between 50 and 200, may result in a faster ordering without a big compromise in quality. The default value for this element of *OPTIONS* is 0.

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

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

This is same as *IPARM(11)* described in Section 6.2.14.

7.1.6 PERM, (type O): permutation vector

```
INTEGER PERM ( N )
```

The description is the same as in Section 6.2.6.

7.1.7 INVP, (type O): inverse permutation vector

```
INTEGER INVP ( N )
```

The description is the same as in Section 6.2.7.

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

```
INTEGER AUX ( NAUX )
```

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

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

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

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

```
INTEGER NAUX
```

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

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

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

WSMSYM (N, IA, JA, PERM, INVP, NNZL, WSPACE, 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 and the integer *WSPACE* contains the number of *double words* of memory required by Cholesky factorization, 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 6.2.14.

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

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

WSMALZ (N, IA, JA, OPTIONS, PERM, INVP, NNZL, WSPACE, AUX, NAUX, 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* [5]. Note that *NAUX* for this routine is the number of *integers* that *AUX* can hold.

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

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 6.2.14.

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

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

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

These routines perform LL^T factorization, assumes that the input matrix is *not* permuted, and uses its permutation 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. 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 6.2.14.

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

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

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

These routines perform LDL^T factorization, assumes that the input matrix is *not* permuted, and uses its permutation 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 *WSxLDL* routine. 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 6.2.14.

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

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

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

These routines perform LDL^T factorization, 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. The double precision output *RCOND* is the inverse of the condition number estimate of the input matrix. 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 6.2.14.

7.7 WSSLV (solve a system using a prior factorization)

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

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. Note that if the user supplied *AUX* in factorization performed earlier (i.e., *NAUX* $>$ 0 in the call to the factorization routine) then the same *AUX* must be passed unaltered to *WSSLV*. If *NAUX* was 0 in the earlier factorization call, then *WSSLV* uses the internally stored copy of the factor matrix. *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. Once again, the the parallel solver is in 0-master mode iff $N = N_0$ and in peer mode iff $N > N_0$.

Figure 3 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 2 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 6.1 and 6.2.2–5 for more details) in Figure 2 for the serial/multithreaded version is valid for the 0-master mode in the parallel version.

The example program in Appendix C illustrates the use of the *PWSSMP* subroutine in 0-master mode for the matrix shown in Figure 2. Appendix E shows the use of *PWSSMP* in the peer mode for the distribution shown in Figure 3.

As shown in Figure 3, 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 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 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 $(0 \le i < p)$:

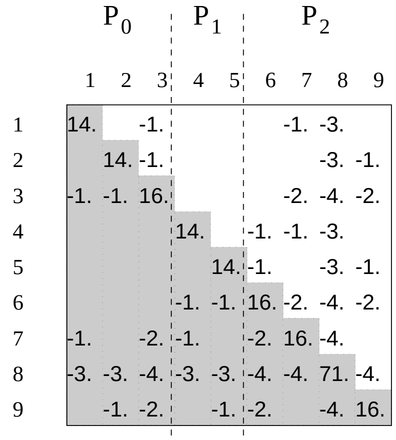
PWSSMP (N$_i$, IA$_i$, JA$_i$, AVALS$_i$, DIAG$_i$, PERM, INVP, B$_i$, LDB$_i$, NRHS, AUX$_i$, NAUX$_i$, MRP$_i$, IPARM, DPARM)

In the 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 6.2; this section is intended to highlight the differences between the serial/multithreaded and the message-passing versions, wherever applicable.

- $\mathbf{N}_i$**:** The number of columns/rows of the matrix A and the number of rows of the right-hand side b residing on 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



A 9 X 9 symmetric sparse matrix.

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

	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					

Figure 3: A sample distribution of the matrix in the two input formats for the peer mode of the message-passing parallel version.

factorization, then user is allowed to use a fresh distribution *only once* before the first Cholesky factorization for a given matrix. Please refer to the description of *IPARM(8)* below for more details.

- $\mathbf{IA}_i$**:** Integer array of size $N_i + 1$. This array provides pointers into the array of indices *JA*. See Figure 3 for more details. If $N_i = 0$, then this parameter can be a *NULL* pointer.
- $\mathbf{JA}_i$**:** Integer array of size *IA*$_i$*(*N_i *+ IPARM(5))* - *IPARM(5)* that contains the global row (column) indices of each column (row) on process P_i. If $N_i = 0$, then this parameter can be a *NULL* pointer.
- $\mathbf{AVALS}_i$**:** Double precision array of size *IA*$_i$*(*N_i *+ IPARM(5))* - *IPARM(5)* that contains the numerical values corresponding to the indices in *JA*$_i$. If $N_i = 0$, then this parameter can be a *NULL* pointer.
- $\mathbf{DIAG}_i$**:** Double precision array of size N_i (only relevant if *IPARM(4)* = 1; i.e., MSR/MSC input format is used). If $N_i = 0$, then this parameter can be a *NULL* pointer.
- **PERM** ***and*** **INVP:** These two integer arrays contain the permutation and the inverse permutation vectors. 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 is 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.*

- $\mathbf{B}_i$**,** $\mathbf{LDB}_i$**,** ***and*** **NRHS:** B is a double precision array of size *LDB*$_i\times$ *NRHS*, where *LDB*$_i \geq N_i$. If $N_i = 0$, then B can be a *NULL* pointer.
- $\mathbf{AUX}_i$ ***and*** $\mathbf{NAUX}_i$**:** If *NAUX*$_i > 0$, then *AUX*$_i$ is a double precision array of size *NAUX*$_i$. If the user wishes to provide space in *AUX*$_i$ for factorization, then the recommended minimum *NAUX*$_i$ is returned in *IPARM(23)* on each process (a different value, in general) after symbolic factorization. Note that *PWSSMP* allocates storage dynamically in addition to that provided in *AUX*$_i$. If space is provided in *AUX*$_i$ for factorization, then the same *AUX*$_i$ must be passed unaltered to *PWSSMP* for triangular solves and iterative refinement.
- $\mathbf{MRP}_i$**:** Integer array of size N_i used only if *IPARM(11)* = 2. If $N_i = 0$, then *MRP* can be a *NULL* pointer.
- **IPARM** ***and*** **DPARM:** The description of *IPARM* and *DPARM* is contained in Sections 6.2.14 and 6.2.15, 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 (e.g., in an interior-point code) and the cost of generating a permuted or unpermuted matrix is more or less the same. In the parallel version, providing a permuted matrix can significantly improve factorization performance by improving the locality in the input data. If *IPARM(8)* is 1 for factorization in the peer mode, then the user is permitted to make the call to *PWSSMP* with different values of N_i than the ones used in previous *PWSSMP* calls for ordering and symbolic factorization. This is because permutation can change the distribution of the matrix. However, after the first *PWSSMP* call for factorization, the values of N_i should not change in any subsequent calls for factorization, triangular solves, or iterative refinement for the same system. This flexibility of redistributing the matrix does not allow the user to change from 0-master mode to peer mode, or vice-versa. If you are using the *WSETGLOBIND* routine (Section 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

The SPINLOOPTIME and YIELDLOOPTIME environment variables are critical to getting good performance on shared-memory systems. Generally recommended values are YIELDLOOPTIME=500 and SPINLOOPTIME=500.

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. Processor P_0 also bears a disproportionately high workload in the ordering phase compared to the other processes. SP installations are often heterogeneous with nodes varying in CPU's as well as the amount of memory on them. On a heterogeneous machine, the performance of the parallel routines will be much better if process P_0 has a fast CPU, high memory bandwidth, and plentiful RAM. The time spent in ordering and symbolic factorization relative to that in the numerical computation phases increases as the number of 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 *IAPRM(27)*. See Sections 6.2.14 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 Node 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 nodes (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.

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

9 Parallel Subroutines Providing a Simpler Interface

In this section, we list the routines and that provide a simpler interface to *PWSSMP*. This interface is analogous to the simple interface to *WSSMP* described in Section 7. Parallel routines *PWSxALZ, 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 Appendix D illustrates the use of the simple parallel interface for the matrix shown in Figure 2. Appendix F gives the source code of many of the simple subroutines to show how they are implemented via calls to *PWSSMP*.

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 and data distribution.

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

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

10.2 *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 $0 \ldots 63$. 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 $0 \ldots 63$, at most 64 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 $0 \ldots 63$, 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 10.3) before another context can be stored with the same *ID*.

After a successful call to *WSTOREMAT*, the current context is destroyed; i.e., either the user should recall a previously stored context (see Section 10.3), or start a new one with the call to ordering or symbolic factorization with a valid input in *PERM, INVP*.

10.3 *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 $0 \ldots 63$. 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* 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 $0 \ldots 63$.

10.4 *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, if any, are also destroyed by a call to *WSMP_CLEAR()* or *PWSMP_CLEAR()*.

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

10.5 *WSFFREE ()*^{S} 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.

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

This routine can be used to sort the rows indices of each column or the column indices or each row (depending on the type of storage) of an $M \times N$ sparse matrix. The size of *IA* is $M + 1$ and the range of indices in *JA* is 0 to $N - 1$ or 1 to N. Only JA is modified upon successful completion, which is indicated by a return value of 0 in *INFO*. The descriptions of *IA* and *JA* are similar to those in Section 6.2. The description of *INFO* is similar to that of *IPRAM(64)*.

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

This routine is similar to *WS_SORTINDICES_I*, except that it also moves the values in *AVALS* according to the sorting of indices in *JA*. The descriptions of *IA*, *JA*, and *AVALS* are similar to those in Section 6.2. The description of *INFO* is similar to that of *IPRAM(64)*.

10.8 *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. Note that *PWSMP* uses the FORTRAN interface of MPI, and therefore, the communicator *INPCOMM* must be generated using MPI's FORTRAN API.

10.9 *WSETMAXTHRDS (MAXTHRD)*^{P,T}

If the underlying hardware is an SMP platform, then *WSMP* and *PWSMP* automatically spawn threads to utilize all the available CPU's. The number of threads spawned is a function of the the number of CPU's on the given node or workstation. The routine *WSETMAXTHRDS* can be used to set the maximum number of threads that would be spawned to the input integer *MAXTHRD*. For example, if you are running four processes on the same node that has four CPU's, you may want to set the maximum number of threads that each process uses to 1 in order to reduce the overheads. If you do not set the maximum number of threads to 1 in this scenario, then several (many more than four) threads will be competing for only 4 CPU's.

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

This routine is useful when you have a parallel job that uses only message-passing parallelism, but your SP has SMP nodes. In this situation, you want to spawn as many processes as the number of CPU's, not the number of nodes. However, you should call *WSETMAXTHRDS* with *MAXTHRD* = 1, so that parallel *WSSMP/PWSSSMP* call on the same node does not spawn multiple threads. Depending on the POE (Parallel Operating Environment) version in use, you may need to set the MP_EUILIB environment variable to *ip* in order to run multiple MPI processes on a node. However, if your POE supports multiple processes per node with MP_EUILIB set to *us*, this setting will yield much better performance.

10.10 *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 3). 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.

A call to *WSETGLOBIND* affects only the current context.

- 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 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. C-style numbering is not permitted with the simple routines of Section 9.
- *GLI*$_i$ is an input *integer* array of size N_i. The k-th element of this integer array contains the global index of the k-th index on process P_j. The default global index corresponding to the k-th index on P_j is $k + \Sigma_{i=0}^{j-1}$. The purpose of using the *WSETGLOBIND* subroutine is to override these defaults.

For example, refer to the matrix shown in Figure 3. 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.

10.11 *WSETLF* (RLF_i)^P

While working in a environment in which the SP nodes are being shared with other users, a *PWSMP* user can specify the load factor on each node using the *WSETLF* routine. The local input argument RLF_i is of type *default real* (currently 4 bytes) and specifies the expected load factor on node P_i while the *PWSMP* code is running. For example if the *PWSMP* user is sharing the first two nodes of a four node ensemble with another user, then RLF_0 and RLF_1 should be 2.0. This routine need not be called on all or any processes. The default value on all processes is initially set to 1.0. The routine must be called before symbolic factorization to be affective, and once called, affects all subsequent calls to *PWSMP* routines for factorization unless RLF_i is set back to the original value by another call to *WSETLF*. In a shared environment, this routine can help *PWSSMP* perform a better load distribution for more effective use of the machine. Needless to say that calling *WSETLF* with the same value on all the nodes has no effect because it is the relative loads that matter. Only values between 0.1 and 10.0 are valid for RLF_i.

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

12 Acknowledgements

We would like to thank Mahesh Joshi of IBM T.J. Watson Research Center for contributing the message-passing parallel triangular solver for symmetric matrices to *PWSMP*.

References

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

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

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

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

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

[6] Anshul Gupta. WSMP: Watson sparse matrix package (Part-II: direct solution of general sparse systems). Technical Report RC 21888 (98472), IBM T. J. Watson Research Center, Yorktown Heights, NY, November 20, 2000.

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

[8] Anshul Gupta, Mahesh Joshi, and Vipin Kumar. WSSMP: A high-performance serial and parallel symmetric sparse linear solver. In Bo Kagstrom, Jack J. Dongarra, Eric Elmroth, and Jerzy Wasniewski, editors, *PARA'98 Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems (Lecture Notes in Computer Science, No. 1541)*. Springer Verlag, June 1998.

[9] 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.

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

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

[12] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly and Associates, Inc., Sebastopol, California, 1996.

[13] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The complete reference*. The MIT Press, Cambridge, Massachusetts, 1996.

13 Appendix A: wssmp_ex1.f

13.1 A serial example program showing the use of *wssmp*

```
C------------------------------------------------------------------------------
C       Example program to show the use of the "wssmp" routine.
C------------------------------------------------------------------------------
C.. This program can be obtained via anonymous FTP from the following site:
C
C   ftp://ftp.cs.umn.edu/users/kumar/anshul/wssmp_ex1.f
C
C   (C) IBM Corporation, 1999.
C
c Acceptance and use of this program constitutes the user's understanding
c that he/she will have no recourse to IBM for any actual or consequential
c damages, including, but not limited to, lost profits or savings, arising
c out of the use or inability to use this program.
C------------------------------------------------------------------------------
        program wssmp_ex1
        implicit none

        integer n, ldb, nrhs, naux
        integer iparm(64)
        integer ia(10)
        integer ja(29)
        integer perm(9), invp(9)
        double precision dparm(64)
        double precision avals(29)
        double precision b(9)

        integer mrp                     ! just a placeholder in this program
        double precision aux, diag      ! just placeholders in this program

        integer i
        double precision waltime, rtc

C.. Fill all arrays containing matrix data.

        data n /9/, ldb /9/, nrhs /1/, naux /0/

        data ia /1,5,9,13,17,21,25,27,29,30/

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

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

        waltime = rtc()

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

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

        iparm(1) = 0
        iparm(2) = 0
        iparm(3) = 0
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.. Ordering.

        waltime = rtc()
        iparm(2) = 1
        iparm(3) = 1
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

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

C.. Symbolic Factorization.

        waltime = rtc()
        iparm(2) = 2
        iparm(3) = 2
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

        print *,'Number of nonzeros in factor L = ',iparm(24)
        print *,'Number of FLOPS in factorization = ',dparm(23)
        print *,'Double words needed to factor = ',iparm(23)

C.. Cholesky Factorization.

        waltime = rtc()
        iparm(2) = 3
        iparm(3) = 3
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
```

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

C.... If Cholesky factorization failed due to non-positive-definite
c     matrix, then try LDL' factorization.

          if (iparm(64) .gt. 0) go to 1000
          stop
        end if

C.. Back substitution.

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

        waltime = rtc()
        iparm(2) = 4
        iparm(3) = 4
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.. Iterative refinement.

        waltime = rtc()
        iparm(2) = 5
        iparm(3) = 5
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

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

C.. Solve the same system using LDL' factorization.

1000 continue

        waltime = rtc()
        iparm(2) = 3
        iparm(3) = 3
        iparm(31) = 1
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
```

```
        waltime = rtc()-waltime
        print *,'LDL^T factorization complete in time - ',waltime
        print *,'Factorization MegaFlops = ',
     +          (dparm(23) * 1.d-6) / waltime
        if (iparm(64) .ne. 0) then
          print *,'The following ERROR was detected: ',iparm(64)
          stop
        end if

C.. Back substitution.

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

        waltime = rtc()
        iparm(2) = 4
        iparm(3) = 4
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.. Iterative refinement.

        waltime = rtc()
        iparm(2) = 5
        iparm(3) = 5
        call wssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

        print *,'Iterative refinement done in time - ',rtc()-waltime
        if (iparm(64) .ne. 0) then
          print *,'The following ERROR was detected: ',iparm(64)
          stop
        end if
        print *,'The solution of the system is as follows:'
        do i = 1, n
          print *,i,' : ',b(i)
        end do

        stop
        end
```

13.2 A script showing the execution of program *wssmp_ex1*

```
% xlf_r -O3 -qstrict -lwsmp -lpthreads -lm_r wssmp_ex1.f
** wssmp_ex1   === End of Compilation 1 ===
1501-510  Compilation successful for file wssmp_ex1.f.
% setenv SPINLOOPTIME 500
% setenv YIELDLOOPTIME 500
% a.out
 Initialization complete in time -  0.184826850891113281E-01
 Ordering complete in time -  0.114488601684570312E-02
 Symbolic complete in time -  0.349402427673339844E-03
 Number of nonzeros in factor L =  30
 Number of FLOPS in factorization =  110.000000000000000
```

```
 Double words needed to factor =  49
 Cholesky complete in time -  0.131613016128540039E-01
 Factorization MegaFlops =  0.835783596757393166E-02
 Back substitution complete in time -  0.125885009765625000E-02
 Iterative refinement done in time -  0.126373767852783203E-02
 The solution of the system is as follows:
 1  :  0.100948241801659427
 2  :  0.100948241801659441
 3  :  0.119517977084156457
 4  :  0.100948241801659427
 5  :  0.100948241801659441
 6  :  0.119517977084156443
 7  :  0.119517977084156443
 8  :  0.580798103516396713E-01
 9  :  0.119517977084156457
 LDL^T factorization complete in time -  0.772237777709960938E-03
 Factorization MegaFlops =  0.142443173819079949
 Back substitution complete in time -  0.870823860168457031E-03
 Iterative refinement done in time -  0.559687614440917969E-03
 The solution of the system is as follows:
 1  :  0.100948241801659414
 2  :  0.100948241801659427
 3  :  0.119517977084156457
 4  :  0.100948241801659427
 5  :  0.100948241801659427
 6  :  0.119517977084156457
 7  :  0.119517977084156457
 8  :  0.580798103516396713E-01
 9  :  0.119517977084156457
%
```

14 Appendix B: wssmp_ex2.f

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

```
C-----------------------------------------------------------------------------
C       Example program to show the use of the simple interface for
c       solving symmetric sparse systems in the WSMP library.
C-----------------------------------------------------------------------------
C.. This program can be obtained via anonymous FTP from the following site:
C
C   ftp://ftp.cs.umn.edu/users/kumar/anshul/wssmp_ex2.f
C
C   (C) IBM Corporation, 1999.
C
c Acceptance and use of this program constitutes the user's understanding
c that he/she will have no recourse to IBM for any actual or consequential
c damages, including, but not limited to, lost profits or savings, arising
c out of the use or inability to use this program.
C-----------------------------------------------------------------------------
        program wssmp_ex2
        implicit none

        integer n, ldb, nrhs, naux, nnzl, wspace, niter, info
        integer options(5)
        integer ia(10)
        integer ja(29)
        integer perm(9), invp(9)
        double precision avals(29)
        double precision b(9)
        double precision rcond

        double precision aux            ! just a placeholder in this program

        integer i
        double precision waltime, rtc

C.. Fill all arrays containing matrix data.

        data n /9/, ldb /9/, nrhs /1/, naux /0/

        data ia /1,5,9,13,17,21,25,27,29,30/

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

```
      data options /5 * 0/

C.. Ordering and Symbolic.

      waltime = rtc()
      call wscalz (n, ia, ja, options, perm, invp, nnzl, wspace,
     +             aux, naux, info)

      print *,'Analyze step complete in time - ',rtc()-waltime
      if (info .ne. 0) then
        print *,'The following ERROR was detected: ',info
        return
      end if
      print *,'Number of nonzeros in factor L = ',nnzl
      print *,'Double words needed to factor = ',wspace

C.. Cholesky Factorization.

      waltime = rtc()
      call wscchf (n, ia, ja, avals, perm, invp, aux, naux, info)

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

C.... If Cholesky factorization failed due to non-positive-definite
c     matrix, then try LDL' factorization.

        if (info .gt. 0) go to 1000
        stop
      end if

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      waltime = rtc()
      niter = 1
      call wsslv (n, perm, invp, b, ldb, nrhs, niter, aux, naux)

      print *,'Back substitution complete in time - ',rtc()-waltime
      print *,'The solution of the system is as follows:'
      do i = 1, n
        print *,i,' : ',b(i)
      end do

C.. Solve the same system using LDL' factorization.

1000 continue

      waltime = rtc()
      call wscldl (n, ia, ja, avals, perm, invp, aux, naux, info)

      print *,'LDL^T factorization complete in time - ',rtc()-waltime
      if (info .ne. 0) then
        print *,'The following ERROR was detected: ',info
        stop
      end if

C.. Back substitution.
```

```
        do i = 1, n
          b(i) = 1.d0
        end do
        waltime = rtc()
        niter = 1
        call wsslv (n, perm, invp, b, ldb, nrhs, niter, aux, naux)

        print *,'Back substitution complete in time - ',rtc()-waltime
        print *,'The solution of the system is as follows:'
        do i = 1, n
          print *,i,' : ',b(i)
        end do

C.. Solve the same system using the expert driver.

        waltime = rtc()
        do i = 1, n
          b(i) = 1.d0
        end do
        call wscsvx (n, ia, ja, avals, perm, invp, b, ldb, nrhs,
     +              aux, naux, rcond, info)

        print *,'Run time for expert driver = ',rtc() - waltime
        if (info .ne. 0) then
          print *,'The following ERROR was detected: ',info
          stop
        end if
        print *,'Condition number estimate of matrix = ',1.d0/rcond
        print *,'The solution of the system is as follows:'
        do i = 1, n
          print *,i,' : ',b(i)
        end do

        stop
        end
```

14.2 A script showing the execution of program *wssmp_ex2*

```
% xlf_r -O3 -qstrict -lwsmp -lpthreads -lm_r wssmp_ex2.f
** wssmp_ex2   === End of Compilation 1 ===
1501-510  Compilation successful for file wssmp_ex2.f.
% setenv SPINLOOPTIME 500
% setenv YIELDLOOPTIME 500
% a.out
 Analyze step complete in time -  0.137259960174560547E-01
 Number of nonzeros in factor L =  30
 Double words needed to factor =  49
 Cholesky complete in time -  0.293751955032348633E-01
 Back substitution complete in time -  0.108671188354492188E-02
 The solution of the system is as follows:
 1  :  0.100948241801659427
 2  :  0.100948241801659427
 3  :  0.119517977084156485
 4  :  0.100948241801659427
 5  :  0.100948241801659427
 6  :  0.119517977084156443
 7  :  0.119517977084156471
 8  :  0.580798103516396783E-01
 9  :  0.119517977084156471
 LDL^T factorization complete in time -  0.104676485061645508E-01
 Back substitution complete in time -  0.127971172332763672E-02
```

```
 The solution of the system is as follows:
 1  :  0.100948241801659427
 2  :  0.100948241801659427
 3  :  0.119517977084156485
 4  :  0.100948241801659427
 5  :  0.100948241801659414
 6  :  0.119517977084156443
 7  :  0.119517977084156485
 8  :  0.580798103516396783E-01
 9  :  0.119517977084156471
 Run time for expert driver =  0.383311510086059570E-01
 Condition number estimate of matrix =  11.2319456916059064
 The solution of the system is as follows:
 1  :  0.100948241801659427
 2  :  0.100948241801659427
 3  :  0.119517977084156485
 4  :  0.100948241801659427
 5  :  0.100948241801659414
 6  :  0.119517977084156443
 7  :  0.119517977084156485
 8  :  0.580798103516396783E-01
 9  :  0.119517977084156471
%
```

15 Appendix C: pwssmp_ex1.f

15.1 A parallel example program showing the 0-master mode in *pwssmp*

```
C-------------------------------------------------------------------------------
C       Example parallel program to show the use of the "pwssmp" routine
c       in 0-master mode.
C-------------------------------------------------------------------------------
C.. This program can be obtained via anonymous FTP from the following site:
C
C   ftp://ftp.cs.umn.edu/users/kumar/anshul/pwssmp_ex1.f
C
C   (C) IBM Corporation, 1999.
C
c Acceptance and use of this program constitutes the user's understanding
c that he/she will have no recourse to IBM for any actual or consequential
c damages, including, but not limited to, lost profits or savings, arising
c out of the use or inability to use this program.
C-------------------------------------------------------------------------------

        program pwssmp_ex1
        implicit none
        include 'mpif.h'
        integer mperr, myid

        call mpi_init(mperr)
        if (mperr .ne. 0) then
          print *,'MPI initialization error'
          stop
        end if
        call mpi_comm_rank(MPI_COMM_WORLD,myid,mperr)

        if (myid .eq. 0) then
          call master (myid)
        else
          call slave (myid)
        end if

        call mpi_finalize(mperr)
        stop
        end
C-------------------------------------------------------------------------------
        subroutine master (myid)
        implicit none
        integer myid

C..  This subroutine is called only on node 0.
C..  It is the same as program wssmp_ex1, with all "stop" statements
c    replaced by "return" statements and all calls to 'wssmp' replaced
c    by calls to 'pwssmp'.

        integer n, ldb, nrhs, naux
        integer iparm(64)
        integer ia(10)
        integer ja(29)
        integer perm(9), invp(9)
        double precision dparm(64)
        double precision avals(29)
        double precision b(9)

        integer mrp                     ! just a placeholder in this program
        double precision aux, diag      ! just placeholders in this program
```

```
        integer i
        double precision waltime, rtc

C.. Fill all arrays containing matrix data.

        data n /9/, ldb /9/, nrhs /1/, naux /0/

        data ia /1,5,9,13,17,21,25,27,29,30/

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

        waltime = rtc()

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

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

        iparm(1) = 0
        iparm(2) = 0
        iparm(3) = 0
        call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +               aux, naux, mrp, iparm, dparm)

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

C.. Ordering.

        waltime = rtc()
        iparm(2) = 1
        iparm(3) = 1
        call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +               aux, naux, mrp, iparm, dparm)

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

```
      end if

C.. Symbolic Factorization.

      waltime = rtc()
      iparm(2) = 2
      iparm(3) = 2
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      print *,'Symbolic complete in time - ',rtc()-waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        return
      end if
      print *,'Number of nonzeros in factor L = ',iparm(24)
      print *,'Number of FLOPS in factorization = ',dparm(23)
      print *,'Double words needed to factor on',myid,'=',iparm(23)

C.. Cholesky Factorization.

      waltime = rtc()
      iparm(2) = 3
      iparm(3) = 3
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.... If Cholesky factorization failed due to non-positive-definite
c     matrix, then try LDL' factorization.

        if (iparm(64) .gt. 0) go to 1000
        return
      end if

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      waltime = rtc()
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.. Iterative refinement.

      waltime = rtc()
      iparm(2) = 5
      iparm(3) = 5
```

```
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      print *,'Iterative refinement done in time - ',rtc()-waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        return
      end if
      print *,'The solution of the system is as follows:'
      do i = 1, n
        print *,i,' : ',b(i)
      end do

C.. Solve the same system using LDL' factorization.

1000 continue

      waltime = rtc()
      iparm(2) = 3
      iparm(3) = 3
      iparm(31) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      waltime = rtc()-waltime
      print *,'LDL^T factorization complete in time - ',waltime
      print *,'Factorization MegaFlops = ',
     +         (dparm(23) * 1.d-6) / waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        return
      end if

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      waltime = rtc()
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.. Iterative refinement.

      waltime = rtc()
      iparm(2) = 5
      iparm(3) = 5
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

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

      return
      end
C----------------------------------------------------------------------------
      subroutine slave (myid)
      implicit none
      integer myid

      integer n, ldb, nrhs, naux
      integer iparm(64)
      integer ia, ja
      integer perm(9), invp(9)
      double precision dparm(64)
      double precision avals, b       ! just placeholders in this routine

      integer mrp                     ! just a placeholder in this program
      double precision aux, diag      ! just placeholders in this program

      data n /0/, ldb /1/, nrhs /1/, naux /0/

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

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Ordering.

      iparm(2) = 1
      iparm(3) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Symbolic Factorization.

      iparm(2) = 2
      iparm(3) = 2
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      print *,'Double words needed to factor on',myid,'=',iparm(23)
      if (iparm(64) .ne. 0) return

C.. Cholesky Factorization.

      iparm(2) = 3
      iparm(3) = 3
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      if (iparm(64) .ne. 0) then
        if (iparm(64) .gt. 0) go to 1000
        return
      end if
```

```
C.. Back substitution.

        iparm(2) = 4
        iparm(3) = 4
        call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +              aux, naux, mrp, iparm, dparm)
        if (iparm(64) .ne. 0) return

C.. Iterative refinement.

        iparm(2) = 5
        iparm(3) = 5
        call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +              aux, naux, mrp, iparm, dparm)
        if (iparm(64) .ne. 0) return

C.. Solve the same system using LDL' factorization.

1000    continue

        iparm(2) = 3
        iparm(3) = 3
        iparm(31) = 1
        call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +              aux, naux, mrp, iparm, dparm)
        if (iparm(64) .ne. 0) return

C.. Back substitution.

        iparm(2) = 4
        iparm(3) = 4
        call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +              aux, naux, mrp, iparm, dparm)
        if (iparm(64) .ne. 0) return

C.. Iterative refinement.

        iparm(2) = 5
        iparm(3) = 5
        call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +              aux, naux, mrp, iparm, dparm)

        return
        end
```

15.2 A script showing the execution of program *pwssmp_ex1*

```
$ export MP_EUILIB=us
$ export MP_EUIDEVICE=css0
$ export MP_CSS_INTERRUPT=no
$ export MP_PULSE=0
$ export MP_RESD=yes
$ export MP_LABELIO=yes
$ export MP_STDOUTMODE=ordered
$ export SPINLOOPTIME=500
$ export YIELDLOOPTIME=500
$ mpxlf_r -O3 -qstrict -lwsmp -lpwsmp -lpthreads -lm_r pwssmp_ex1.f
** pwssmp_ex1   === End of Compilation 1 ===
** master   === End of Compilation 2 ===
** slave   === End of Compilation 3 ===
1501-510  Compilation successful for file pwssmp_ex1.f.
```

```
$ poe a.out -procs 3
   0: Initialization complete in time -  0.400943636894226074
   0: Ordering complete in time -  0.407461166381835938
   0: Symbolic complete in time -  0.237387180328369141
   0: Number of nonzeros in factor L =  35
   0: Number of FLOPS in factorization =  159.000000000000000
   0: Double words needed to factor on 0 = 47
   0: Cholesky complete in time -  0.330920696258544922
   0: Factorization MegaFlops =  0.480477654609353690E-03
   0: Back substitution complete in time -  0.699319839477539062E-01
   0: Iterative refinement done in time -  0.821232795715332031E-03
   0: The solution of the system is as follows:
   0: 1  :  0.100948241801659427
   0: 2  :  0.100948241801659427
   0: 3  :  0.119517977084156457
   0: 4  :  0.100948241801659427
   0: 5  :  0.100948241801659427
   0: 6  :  0.119517977084156457
   0: 7  :  0.119517977084156457
   0: 8  :  0.580798103516396644E-01
   0: 9  :  0.119517977084156471
   0: LDL^T factorization complete in time -  0.145858168601989746
   0: Factorization MegaFlops =  0.109010007134993582E-02
   0: Back substitution complete in time -  0.686526298522949219E-03
   0: Iterative refinement done in time -  0.741600990295410156E-03
   0: The solution of the system is as follows:
   0: 1  :  0.100948241801659427
   0: 2  :  0.100948241801659427
   0: 3  :  0.119517977084156457
   0: 4  :  0.100948241801659427
   0: 5  :  0.100948241801659427
   0: 6  :  0.119517977084156457
   0: 7  :  0.119517977084156471
   0: 8  :  0.580798103516396713E-01
   0: 9  :  0.119517977084156471
   1: Double words needed to factor on 1 = 17
   2: Double words needed to factor on 2 = 0
$
```

16 Appendix D: pwssmp_ex2.f

16.1 A parallel example program showing the use of the simple interface

```
C-----------------------------------------------------------------------------
C       Example parallel program to show the use of the simple interface
c       for solving symmetric systems in the PWSMP library in 0-master mode.
C-----------------------------------------------------------------------------
C.. This program can be obtained via anonymous FTP from the following site:
C
C   ftp://ftp.cs.umn.edu/users/kumar/anshul/pwssmp_ex2.f
C
C   (C) IBM Corporation, 1999.
C
c       Acceptance and use of this program constitutes the user's understanding
c       that he/she will have no recourse to IBM for any actual or consequential
c       damages, including, but not limited to, lost profits or savings, arising
c       out of the use or inability to use this program.
C-----------------------------------------------------------------------------

        program pwssmp_ex1
        implicit none
        include 'mpif.h'
        integer mperr, myid

        call mpi_init(mperr)
        if (mperr .ne. 0) then
          print *,'MPI initialization error'
          stop
        end if
        call mpi_comm_rank(MPI_COMM_WORLD,myid,mperr)

        if (myid .eq. 0) then
          call master (myid)
        else
          call slave (myid)
        end if

        call mpi_finalize(mperr)
        stop
        end
C-----------------------------------------------------------------------------
        subroutine master (myid)
        implicit none
        integer myid

C..  This subroutine is called only on node 0.
C..  It is very similar to program wssmp_ex2.

        integer n, ldb, nrhs, naux, nnzl, wspace, niter, info
        integer options(5)
        integer ia(10)
        integer ja(29)
        integer perm(9), invp(9)
        double precision avals(29)
        double precision b(9)
        double precision rcond

        double precision aux ! just placeholders in this program

        integer i
        double precision waltime, rtc
```

```
C.. Fill all arrays containing matrix data.

        data n /9/, ldb /9/, nrhs /1/, naux /0/

        data ia /1,5,9,13,17,21,25,27,29,30/

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

        data options /5 * 0/

C.. Ordering and Symbolic.

        waltime = rtc()
        call pwscalz (n, ia, ja, options, perm, invp, nnzl, wspace,
     +                aux, naux, info)

        print *,'Analyze step complete in time - ',rtc()-waltime
        if (info .ne. 0) then
          print *,'The following ERROR was detected: ',info
          return
        end if
        print *,'Number of nonzeros in factor L = ',nnzl
        print *,'Double words needed to factor on',myid,'=',wspace

C.. Cholesky Factorization.

        waltime = rtc()
        call pwscchf (n, ia, ja, avals, perm, invp, aux, naux, info)

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

C.... If Cholesky factorization failed due to non-positive-definite
c     matrix, then try LDL' factorization.

          if (info .gt. 0) go to 1000
          return
        end if

C.. Back substitution.

        do i = 1, n
```

```
          b(i) = 1.d0
        end do
        waltime = rtc()
        niter = 1
        call pwsslv (n, perm, invp, b, ldb, nrhs, niter, aux, naux)

        print *,'Back substitution complete in time - ',rtc()-waltime
        print *,'The solution of the system is as follows:'
        do i = 1, n
          print *,i,' : ',b(i)
        end do

C.. Solve the same system using LDL' factorization.

1000    continue

        waltime = rtc()
        call pwscldl (n, ia, ja, avals, perm, invp, aux, naux, info)

        print *,'LDL^T factorization complete in time - ',rtc()-waltime
        if (info .ne. 0) then
          print *,'The following ERROR was detected: ',info
          return
        end if

C.. Back substitution.

        do i = 1, n
          b(i) = 1.d0
        end do
        waltime = rtc()
        niter = 1
        call pwsslv (n, perm, invp, b, ldb, nrhs, niter, aux, naux)

        print *,'Back substitution complete in time - ',rtc()-waltime
        print *,'The solution of the system is as follows:'
        do i = 1, n
          print *,i,' : ',b(i)
        end do

C.. Solve the same system using the expert driver.

        waltime = rtc()
        do i = 1, n
          b(i) = 1.d0
        end do
        call pwscsvx (n, ia, ja, avals, perm, invp, b, ldb, nrhs,
     +                aux, naux, rcond, info)

        print *,'Run time for expert driver = ',rtc() - waltime
        if (info .ne. 0) then
          print *,'The following ERROR was detected: ',info
          return
        end if
        print *,'The solution of the system is as follows:'
        do i = 1, n
          print *,i,' : ',b(i)
        end do

        return
        end
C-----------------------------------------------------------------------------
        subroutine slave (myid)
```

```
      implicit none
      integer myid

      integer n, ldb, nrhs, naux, niter, wspace, nnzl, info
      integer ia, ja
      integer options(5)
      integer perm(9), invp(9)
      double precision rcond
      double precision avals, b       ! just placeholders in this routine

      double precision aux ! just placeholders in this program

      data n /0/, ldb /1/, nrhs /1/, naux /0/
      data options /5 * 0/

C.. Ordering and Symbolic.

      call pwscalz (n, ia, ja, options, perm, invp, nnzl, wspace,
     +              aux, naux, info)
if (info .ne. 0) return
      print *,'Double words needed to factor on',myid,'=',wspace

C.. Cholesky Factorization.

      call pwscchf (n, ia, ja, avals, perm, invp, aux, naux, info)

      if (info .ne. 0) then

C.... If Cholesky factorization failed due to non-positive-definite
c     matrix, then try LDL' factorization.

        if (info .gt. 0) go to 1000
        return
      end if

C.. Back substitution.

      niter = 1
      call pwsslv (n, perm, invp, b, ldb, nrhs, niter, aux, naux)

C.. Solve the same system using LDL' factorization.

1000  continue

      call pwscldl (n, ia, ja, avals, perm, invp, aux, naux, info)
      if (info .ne. 0) return

C.. Back substitution.

      niter = 1
      call pwsslv (n, perm, invp, b, ldb, nrhs, niter, aux, naux)

C.. Solve the same system using the expert driver.

      call pwscsvx (n, ia, ja, avals, perm, invp, b, ldb, nrhs,
     +              aux, naux, rcond, info)
      if (info .ne. 0) return

      return
      end
```

16.2 A script showing the execution of program *pwssmp_ex2*

```
$ export MP_EUILIB=us
$ export MP_EUIDEVICE=css0
$ export MP_CSS_INTERRUPT=no
$ export MP_PULSE=0
$ export MP_RESD=yes
$ export MP_LABELIO=yes
$ export MP_STDOUTMODE=ordered
$ export SPINLOOPTIME=500
$ export YIELDLOOPTIME=500
$ mpxlf_r -O3 -qstrict -lwsmp -lpwsmp -lpthreads -lm_r pwssmp_ex2.f
** pwssmp_ex1   === End of Compilation 1 ===
** master   === End of Compilation 2 ===
** slave   === End of Compilation 3 ===
1501-510  Compilation successful for file pwssmp_ex2.f.
$ poe a.out -procs 3
   0: Analyze step complete in time -  0.957372069358825684
   0: Number of nonzeros in factor L =  35
   0: Double words needed to factor on 0 = 47
   0: Cholesky complete in time -  0.222006797790527344
   0: Back substitution complete in time -  0.110468745231628418
   0: The solution of the system is as follows:
   0: 1  :  0.100948241801659427
   0: 2  :  0.100948241801659427
   0: 3  :  0.119517977084156471
   0: 4  :  0.100948241801659427
   0: 5  :  0.100948241801659427
   0: 6  :  0.119517977084156443
   0: 7  :  0.119517977084156457
   0: 8  :  0.580798103516396505E-01
   0: 9  :  0.119517977084156471
   0: LDL^T factorization complete in time -  0.148649930953979492
   0: Back substitution complete in time -  0.244580507278442383E-01
   0: The solution of the system is as follows:
   0: 1  :  0.100948241801659427
   0: 2  :  0.100948241801659414
   0: 3  :  0.119517977084156471
   0: 4  :  0.100948241801659414
   0: 5  :  0.100948241801659400
   0: 6  :  0.119517977084156429
   0: 7  :  0.119517977084156471
   0: 8  :  0.580798103516396644E-01
   0: 9  :  0.119517977084156443
   0: Run time for expert driver =  0.222575664520263672E-02
   0: The solution of the system is as follows:
   0: 1  :  0.100948241801659427
   0: 2  :  0.100948241801659414
   0: 3  :  0.119517977084156471
   0: 4  :  0.100948241801659414
   0: 5  :  0.100948241801659400
   0: 6  :  0.119517977084156429
   0: 7  :  0.119517977084156471
   0: 8  :  0.580798103516396644E-01
   0: 9  :  0.119517977084156443
   1: Double words needed to factor on 1 = 17
   2: Double words needed to factor on 2 = 0
$
```

17 Appendix E: pwssmp_ex3.f

17.1 A parallel example program showing the peer mode in *pwssmp*

```
C-----------------------------------------------------------------------------
C       Example parallel program to show the use of the "pwssmp" routine
c       in the peer mode.
C-----------------------------------------------------------------------------
C.. This program can be obtained via anonymous FTP from the following site:
C
C   ftp://ftp.cs.umn.edu/users/kumar/anshul/pwssmp_ex3.f
C
C   (C) IBM Corporation, 1999.
C
c Acceptance and use of this program constitutes the user's understanding
c that he/she will have no recourse to IBM for any actual or consequential
c damages, including, but not limited to, lost profits or savings, arising
c out of the use or inability to use this program.
C-----------------------------------------------------------------------------

        program pwssmp_ex1
        implicit none
        include 'mpif.h'
        integer mperr, myid

        call mpi_init(mperr)
        if (mperr .ne. 0) then
          print *,'MPI initialization error'
          stop
        end if
        call mpi_comm_rank(MPI_COMM_WORLD,myid,mperr)

C..  This program must be called on *at least* 3 nodes.

        if (myid .eq. 0) then
          call master0
        else if (myid .eq. 1) then
          call master1
        else if (myid .eq. 2) then
          call master2
        else
          call slave (myid)
        end if

        call mpi_finalize(mperr)
        stop
        end
C-----------------------------------------------------------------------------
        subroutine master0
        implicit none

        integer n, ldb, nrhs, naux
        integer iparm(64)
        integer ia(4)
        integer ja(12)
        integer perm(9), invp(9)
        double precision dparm(64)
        double precision avals(12)
        double precision b(3)

        integer mrp                     ! just a placeholder in this program
        double precision aux, diag      ! just placeholders in this program
```

```
      integer i
      double precision waltime, rtc

C.. Fill all arrays containing matrix data.

      data n /3/, ldb /3/, nrhs /1/, naux /0/

      data ia /1,5,9,13/

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

C.. Executable part of the program starts here.

      waltime = rtc()

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

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

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.. Ordering.

      waltime = rtc()
      iparm(2) = 1
      iparm(3) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.. Symbolic Factorization.

      waltime = rtc()
      iparm(2) = 2
      iparm(3) = 2
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
```

```
      print *,'Symbolic complete in time - ',rtc()-waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        return
      end if
      print *,'Number of nonzeros in factor L = ',iparm(24)
      print *,'Number of FLOPS in factorization = ',dparm(23)
      print *,'Double words needed to factor on 0 =',iparm(23)

C.. Cholesky Factorization.

      waltime = rtc()
      iparm(2) = 3
      iparm(3) = 3
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

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

C.... If Cholesky factorization failed due to non-positive-definite
c     matrix, then try LDL' factorization.

        if (iparm(64) .gt. 0) go to 1000
        return
      end if

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      waltime = rtc()
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      print *,'Back substitution complete in time - ',rtc()-waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        return
      end if
      print *,'The solution of the system is as follows:'
      do i = 1, n
        print *,i,' : ',b(i)
      end do

C.. Solve the same system using LDL' factorization.

1000 continue

      waltime = rtc()
      iparm(2) = 3
      iparm(3) = 3
      iparm(31) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
```

```
      waltime = rtc()-waltime
      print *,'LDL^T factorization complete in time - ',waltime
      print *,'Factorization MegaFlops = ',
     +        (dparm(23) * 1.d-6) / waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        return
      end if

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      waltime = rtc()
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +            aux, naux, mrp, iparm, dparm)

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

C.. Iterative refinement.

      waltime = rtc()
      iparm(2) = 5
      iparm(3) = 5
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +            aux, naux, mrp, iparm, dparm)

      print *,'Iterative refinement done in time - ',rtc()-waltime
      if (iparm(64) .ne. 0) then
        print *,'The following ERROR was detected: ',iparm(64)
        return
      end if
      print *,'The solution of the system is as follows:'
      do i = 1, n
        print *,i,' : ',b(i)
      end do

      return
      end
C-------------------------------------------------------------------------------
      subroutine master1
      implicit none

      integer n, ldb, nrhs, naux
      integer iparm(64)
      integer ia(3)
      integer ja(8)
      integer perm(9), invp(9)
      double precision dparm(64)
      double precision avals(8)
      double precision b(2)

      integer mrp                    ! just a placeholder in this program
      double precision aux, diag     ! just placeholders in this program

      integer i
```

```
      double precision waltime, rtc

C.. Fill all arrays containing matrix data.

      data n /2/, ldb /2/, nrhs /1/, naux /0/

      data ia /1,5,9/

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

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

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Ordering.

      iparm(2) = 1
      iparm(3) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Symbolic Factorization.

      iparm(2) = 2
      iparm(3) = 2
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

      print *,'Double words needed to factor on 1 =',iparm(23)

C.. Cholesky Factorization.

      iparm(2) = 3
      iparm(3) = 3
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      if (iparm(64) .ne. 0) then
        if (iparm(64) .gt. 0) go to 1000
        return
      end if

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
```

```
      if (iparm(64) .ne. 0) return

      do i = 1, n
        print *,i+3,' : ',b(i)
      end do

C.. Solve the same system using LDL' factorization.

1000  continue

      iparm(2) = 3
      iparm(3) = 3
      iparm(31) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Iterative refinement.

      iparm(2) = 5
      iparm(3) = 5
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

      do i = 1, n
        print *,i+3,' : ',b(i)
      end do

      return
      end
C------------------------------------------------------------------------------
      subroutine master2
      implicit none

      integer n, ldb, nrhs, naux
      integer iparm(64)
      integer ia(5)
      integer ja(9)
      integer perm(9), invp(9)
      double precision dparm(64)
      double precision avals(9)
      double precision b(4)

      integer mrp                     ! just a placeholder in this program
      double precision aux, diag      ! just placeholders in this program

      integer i
      double precision waltime, rtc

C.. Fill all arrays containing matrix data.
```

```fortran
      data n /4/, ldb /4/, nrhs /1/, naux /0/

      data ia /1,5,7,9,10/

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

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

      iparm(1) = 0
      iparm(2) = 0
      iparm(3) = 0
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Ordering.

      iparm(2) = 1
      iparm(3) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Symbolic Factorization.

      iparm(2) = 2
      iparm(3) = 2
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

      print *,'Double words needed to factor on 2 =',iparm(23)

C.. Cholesky Factorization.

      iparm(2) = 3
      iparm(3) = 3
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      if (iparm(64) .ne. 0) then
        if (iparm(64) .gt. 0) go to 1000
        return
      end if

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
```

```
      if (iparm(64) .ne. 0) return

      do i = 1, n
        print *,i+5,' : ',b(i)
      end do

C.. Solve the same system using LDL' factorization.

1000  continue

      iparm(2) = 3
      iparm(3) = 3
      iparm(31) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Back substitution.

      do i = 1, n
        b(i) = 1.d0
      end do
      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Iterative refinement.

      iparm(2) = 5
      iparm(3) = 5
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

      do i = 1, n
        print *,i+5,' : ',b(i)
      end do

      return
      end
C-----------------------------------------------------------------------------
      subroutine slave (myid)
      implicit none
      integer myid

      integer n, ldb, nrhs, naux
      integer iparm(64)
      integer ia, ja
      integer perm(9), invp(9)
      double precision dparm(64)
      double precision avals, b       ! just placeholders in this routine

      integer mrp                     ! just a placeholder in this program
      double precision aux, diag      ! just placeholders in this program

      data n /0/, ldb /1/, nrhs /1/, naux /0/

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

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

```
      iparm(3) = 0
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
   +              aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Ordering.

      iparm(2) = 1
      iparm(3) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
   +              aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Symbolic Factorization.

      iparm(2) = 2
      iparm(3) = 2
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
   +              aux, naux, mrp, iparm, dparm)
      print *,'Double words needed to factor on',myid,'=',iparm(23)
      if (iparm(64) .ne. 0) return

C.. Cholesky Factorization.

      iparm(2) = 3
      iparm(3) = 3
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
   +              aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) then
        if (iparm(64) .gt. 0) go to 1000
        return
      end if

C.. Back substitution.

      iparm(2) = 4
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
   +              aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Iterative refinement.

      iparm(2) = 5
      iparm(3) = 5
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
   +              aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Solve the same system using LDL' factorization.

1000  continue

      iparm(2) = 3
      iparm(3) = 3
      iparm(31) = 1
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
   +              aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Back substitution.

      iparm(2) = 4
```

```
      iparm(3) = 4
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)
      if (iparm(64) .ne. 0) return

C.. Iterative refinement.

      iparm(2) = 5
      iparm(3) = 5
      call pwssmp (n, ia, ja, avals, diag, perm, invp, b, ldb, nrhs,
     +             aux, naux, mrp, iparm, dparm)

      return
      end
```

17.2 A script showing the execution of program *pwssmp_ex3*

```
$ mpxlf_r -O3 -qstrict -lwsmp -lpwsmp -lpthreads -lm_r pwssmp_ex3.f
** pwssmp_ex1   === End of Compilation 1 ===
** master0   === End of Compilation 2 ===
** master1   === End of Compilation 3 ===
** master2   === End of Compilation 4 ===
** slave   === End of Compilation 5 ===
1501-510  Compilation successful for file pwssmp_ex3.f.
$ export MP_EUILIB=us
$ export MP_EUIDEVICE=css0
$ export MP_CSS_INTERRUPT=no
$ export MP_PULSE=0
$ export MP_RESD=yes
$ export MP_LABELIO=yes
$ export MP_STDOUTMODE=unordered
$ export SPINLOOPTIME=500
$ export YIELDLOOPTIME=500
$ poe a.out -procs 3
   0: Initialization complete in time -  0.145274639129638672
   1: Double words needed to factor on 1 = 17
   2: Double words needed to factor on 2 = 0
   0: Ordering complete in time -  0.289905071258544922E-02
   0: Symbolic complete in time -  0.181317329406738281E-02
   0: Number of nonzeros in factor L =  35
   0: Number of FLOPS in factorization =  159.000000000000000
   0: Double words needed to factor on 0 = 47
   0: Cholesky complete in time -  0.649464130401611328E-02
   0: Factorization MegaFlops =  0.244817215543033338E-01
   0: Back substitution complete in time -  0.685453414916992188E-03
   0: The solution of the system is as follows:
   0: 1  :  0.100948241801659427
   0: 2  :  0.100948241801659427
   0: 3  :  0.119517977084156471
   1: 4  :  0.100948241801659427
   2: 6  :  0.119517977084156443
   1: 5  :  0.100948241801659427
   2: 7  :  0.119517977084156457
   2: 8  :  0.580798103516396505E-01
   2: 9  :  0.119517977084156471
   0: LDL^T factorization complete in time -  0.215065479278564453E-02
   0: Factorization MegaFlops =  0.739309723407793329E-01
   0: Back substitution complete in time -  0.702381134033203125E-03
   0: Iterative refinement done in time -  0.760316848754882812E-03
   0: The solution of the system is as follows:
   1: 4  :  0.100948241801659427
```

```
   1: 5  :  0.100948241801659427
   2: 6  :  0.119517977084156457
   2: 7  :  0.119517977084156471
   2: 8  :  0.580798103516396713E-01
   2: 9  :  0.119517977084156471
   0: 1  :  0.100948241801659427
   0: 2  :  0.100948241801659427
   0: 3  :  0.119517977084156457
$
```

18 Appendix F: The source code of the simple interface

18.1 The include file *wssimp.h*

This include file is used in all serial and parallel simple-interface subroutines and also reflects the default values of *IPARM* and *DPARM* as shown in Table 1.

```
        integer difopts(64)
        double precision ddfopts(64)
        static ddfopts, difopts

C.. These initialize IPRAM and DPARM to default values.

        data difopts   /1,      0,      0,      0,
     1                  1,      1,      0,      0,
     2                  0,      0,      0,      0,
     3                  0,      0,      0,      1,
     4                  0,      1,      0,      0,
     5                  0,      0,      0,      0,
     6                  0,      0,      0,      0,
     7                  0,      0,      0,      0,
     8                  0,      0,      0,      0,
     9                  0,      0,      0,      0,
     A                  0,      0,      0,      0,
     1                  0,      0,      0,      0,
     2                  0,      0,      0,      0,
     3                  0,      0,      0,      0,
     4                  0,      0,      0,      0,
     5                  0,      0,      0,      0/

        data ddfopts   /0.d0,   0.d0,   0.d0,   0.d0,
     1                  0.d0,   1.d-9,  0.d0,   0.d0,
     2                  0.d0,   1.d-02, 1.d-30, 1.d-16,
     3                  0.d0,   0.d0,   0.d0,   0.d0,
     4                  0.d0,   0.d0,   0.d0,   0.d0,
     5                  1.d200, 1.d-8,  0.d0,   0.d0,
     6                  0.d0,   0.d0,   0.d0,   0.d0,
     7                  0.d0,   0.d0,   0.d0,   0.d0,
     8                  0.d0,   0.d0,   0.d0,   0.d0,
     9                  0.d0,   0.d0,   0.d0,   0.d0,
     A                  0.d0,   0.d0,   0.d0,   0.d0,
     1                  0.d0,   0.d0,   0.d0,   0.d0,
     2                  0.d0,   0.d0,   0.d0,   0.d0,
     3                  0.d0,   0.d0,   0.d0,   0.d0,
     4                  0.d0,   0.d0,   0.d0,   0.d0,
     5                  0.d0,   0.d0,   0.d0,   0.d0/
```

18.2 The routines for the serial simple interface with CSC-LT input

```
C-----------------------------------------------------------------------------
      subroutine wscalz(n,ia,ja,options,perm,invp,nnzl,wspace,
     1                  aux,naux,info)
      integer n, nnzl,wspace,ia(*),ja(*),options(5),perm(*),invp(*)
      integer aux(*),naux
      include 'wssimp.h'

      difopts(2) = 1                ! Ordering
      difopts(3) = 2                ! Symbolic
      difopts(16) = options(1)
      difopts(17) = options(2)
```

```
      difopts(18) = options(3)
      difopts(19) = options(4)
      difopts(20) = options(5)

      call wssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1           difopts,ddfopts)
      wspace = difopts(23)
      nnzl = difopts(24)
      info = difopts(64)

      return
      end
C-----------------------------------------------------------------------------
      subroutine wscsym(n,ia,ja,perm,invp,nnzl,wspace,info)
      integer n, nnzl,wspace,info,ia(*),ja(*),perm(*),invp(*)
      include 'wssimp.h'

      difopts(2) = 2                ! Symbolic
      difopts(3) = 2                ! Symbolic

      call wssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1           difopts,ddfopts)
      wspace = difopts(23)
      nnzl = difopts(24)
      info = difopts(64)

      return
      end
C-----------------------------------------------------------------------------
      subroutine wscchf(n,ia,ja,avals,perm,invp,aux,naux,info)
      integer n, ia(*),ja(*),perm(*),invp(*),naux,info
      double precision avals(*),aux(*)
      include 'wssimp.h'

      difopts(2) = 3                ! Cholesky
      difopts(3) = 3                ! Cholesky

      call wssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1           difopts,ddfopts)
      info = difopts(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine wscldl(n,ia,ja,avals,perm,invp,aux,naux,info)
      integer n, ia(*),ja(*),perm(*),invp(*),naux,info
      double precision avals(*),aux(*)
      include 'wssimp.h'

      difopts(2) = 3                ! LDL'
      difopts(3) = 3                ! LDL'
      difopts(31) = 1

      call wssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1           difopts,ddfopts)
      info = difopts(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine wscsvx(n,ia,ja,avals,perm,invp,b,ldb,nrhs,
     1                  aux,naux,rcond,info)
      integer n, ldb,nrhs,info,ia(*),ja(*),perm(*),invp(*),naux
      double precision avals(*),rcond,b(ldb,nrhs),aux(*)
      include 'wssimp.h'
```

```
      difopts(2) = 3                ! LDL'
      difopts(3) = 5                ! Iterative Refinement
      difopts(6) = 2
      difopts(7) = 3
      difopts(10) = 2
      difopts(25) = 3
      difopts(31) = 1

      call wssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1           difopts,ddfopts)
      info = difopts(64)
      rcond = 1.d0/(ddfopts(2)*ddfopts(3))
      return
      end
C-----------------------------------------------------------------------
      subroutine wsslv(n,perm,invp,b,ldb,nrhs,niter,aux,naux)
      integer n,nrhs,naux,niter
      integer perm(*),invp(*),b(ldb,nrhs)
      double precision aux(*)
      include 'wssimp.h'

      difopts(2) = 4                ! Solve
      difopts(3) = 4                ! Solve
      if (niter .gt. 0) then
        difopts(3) = 5
        difopts(6) = niter
        difopts(7) = 7
      end if

      call wssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1           difopts,ddfopts)
      return
      end
C-----------------------------------------------------------------------
```

18.3 The routines for the parallel simple interface with CSC-LT input

```
C------------------------------------------------------------------------
      subroutine pwscalz(n,ia,ja,options,perm,invp,nnzl,wspace,
     1                   aux,naux,info)
      integer n, nnzl,wspace,ia(*),ja(*),options(5),perm(*),invp(*)
      integer aux(*),naux
      include 'wssimp.h'

      difopts(2) = 1
      difopts(3) = 2
      difopts(16) = options(1)
      difopts(17) = options(2)
      difopts(18) = options(3)
      difopts(19) = options(4)
      difopts(20) = options(5)

      call pwssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1            difopts,ddfopts)
      wspace = difopts(23)
      nnzl = difopts(24)
      info = difopts(64)

      return
      end
```

```
C-----------------------------------------------------------------------------
      subroutine pwscsym(n,ia,ja,perm,invp,nnzl,wspace,info)
      integer n, nnzl,wspace,info,ia(*),ja(*),perm(*),invp(*)
      include 'wssimp.h'

      difopts(2) = 2
      difopts(3) = 2

      call pwssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1            difopts,ddfopts)
      wspace = difopts(23)
      nnzl = difopts(24)
      info = difopts(64)

      return
      end
C-----------------------------------------------------------------------------
      subroutine pwscchf(n,ia,ja,avals,perm,invp,aux,naux,info)
      integer n, ia(*),ja(*),perm(*),invp(*),naux,info
      double precision avals(*),aux(*)
      include 'wssimp.h'

      difopts(2) = 3
      difopts(3) = 3

      call pwssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1            difopts,ddfopts)
      info = difopts(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine pwscldl(n,ia,ja,avals,perm,invp,aux,naux,info)
      integer n, ia(*),ja(*),perm(*),invp(*),naux,info
      double precision avals(*),aux(*)
      include 'wssimp.h'

      difopts(2) = 3
      difopts(3) = 3
      difopts(31) = 1

      call pwssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1            difopts,ddfopts)
      info = difopts(64)
      return
      end
C-----------------------------------------------------------------------------
      subroutine pwscsvx(n,ia,ja,avals,perm,invp,b,ldb,nrhs,
     1                   aux,naux,rcond,info)
      integer n, ldb,nrhs,info,ia(*),ja(*),perm(*),invp(*),naux
      double precision avals(*),rcond,b(ldb,nrhs),aux(*)
      include 'wssimp.h'

      difopts(2) = 3
      difopts(3) = 5
      difopts(6) = 2
      difopts(7) = 3
      difopts(31) = 1

      call pwssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1            difopts,ddfopts)
      info = difopts(64)
      rcond = 0.d0
```

```
      return
      end
C-----------------------------------------------------------------------------
      subroutine pwsslv(n,perm,invp,b,ldb,nrhs,niter,aux,naux)
      integer n,nrhs,naux,niter
      integer perm(*),invp(*),b(ldb,nrhs)
      double precision aux(*)
      include 'wssimp.h'

      difopts(2) = 4
      difopts(3) = 4
      if (niter .gt. 0) then
        difopts(3) = 5
        difopts(6) = niter
        difopts(7) = 3
      end if

      call pwssmp(n,ia,ja,avals,diag,perm,invp,b,ldb,nrhs,aux,naux,mrp,
     1            difopts,ddfopts)
      return
      end
C-----------------------------------------------------------------------------
```