RC 21888 (98472) November 20, 2000 (Last update: January 2, 2021)
Computer Science/Mathematics

IBM Research Report

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

Version 20.12

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

Anshul Gupta

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

anshul@us.ibm.com

IBM Research

WSMP: Watson Sparse Matrix Package
Part II – direct solution of general systems
Version 20.12

Anshul Gupta

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

anshul@us.ibm.com

IBM Research Report RC 21888 (98472)

November 20, 2000

©IBM Corporation 1997, 2020. All Rights Reserved.

Contents

1 Introduction to Part II

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

For solving general sparse systems, *WSMP* uses a modified version of the multifrontal algorithm [1, 13] for matrices with an unsymmetric pattern of nonzeros. *WSMP* supports threshold partial pivoting for general matrices with a user defined threshold. Detailed performance results of WSMP and a comparison of various general sparse solver packages can be found in [11]. The serial and distributed-memory parallel algorithms used in *WSMP* for solving general sparse systems are described by Gupta [7, 6]. In a shared-memory environment, the threads are managed through a task-parallel engine [14] that strives to achieve load balance via work-stealing.

Unlike the symmetric solver in Part I, *WSMP*'s general solver does not have out-of-core capabilities and the problems must fit in the main memory for reasonable performance.

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

The functionality and the calling sequences of the serial, multithreaded, and the message-passing parallel versions are almost identical. This document is organized accordingly and the descriptions of most parameters for both versions is included in the description of the combined serial and multithreaded version. The serial version supports certain features that the current message-passing parallel version does not. Such features, options, or data structures supported exclusively by the serial version will be annotated by a superscript ^{S} in this document. Similarly, items relevant only to the multithreaded version appear with a superscript ^{T} and those relevant to the message-passing parallel version appear with a superscript ^{P}.

Note 1.1 *Although* WSMP *library contains multithreaded code, the library itself is* **not** *thread-safe. Therefore, the calling program cannot invoke multiple instances of the routines contained in* WSMP *from different threads at the same time.*

Note 1.2 *The message-passing parallel unsymmetric solver requires MPI_THREAD_MULTIPLE support. Therefore, MPI must be initialized accordingly. If MPI_THREAD_MULTIPLE support is not available, then you can use only one thread per MPI process. This can be accomplished by following the instructions in Section 10.4.*

The organization of this document is as follows. Section 2 describes important recent changes in the software that may affect the users of earlier versions. Section 3 lists the various libraries that are available and describe how to obtain and use the libraries. Section 4 gives an overview of the functionality of *WSMP* for solving general sparse systems. Section 5 gives a detailed description of the main serial/multithreaded routine that provides an advanced single-routine interface to the entire software. This section also describes the input data structures for the serial and multithreaded cases. In this section, the differences from the message-passing parallel version are noted, wherever applicable. Section 6 describes user callable routines that provide a simpler interface to the serial and multithreaded solver, but omit some of the advanced features. Section 7 describes how *WSMP*'s general sparse solver can be used to update a previously performed LU factorization. Section 8 describes the input data structures for the parallel solution and reminds users of the differences between the serial and the message-passing parallel versions, wherever applicable. This section does not repeat the information contained in Section 5 because the two user-interfaces are quite similar. Section 9 is the parallel analog of Section 6 and describes user callable routines that provide a simpler interface to the message-passing parallel solver. Section 10 describes a few utility routines available to the users. Section 11 gives a brief description of the double-complex data type interface of *WSMP*'s unsymmetric direct solvers. Section 12 contains the terms and conditions that all users of the package must adhere to.

2 Recent Changes and Other Important Notes

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

Iterative solvers preconditioned with incomplete LU factorization, with or without pivoting, are now available. Please refer to the documentation for Part III, which can be found at *http://www.research.ibm.com/projects/wsmp*.

3 Obtaining, Linking, and Running *WSMP*

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

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

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

3.1 Libraries and other system requirements

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

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

3.2 License file

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

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

3.3 Linking on various systems

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

3.3.1 Linux on x86_64 platforms

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.2 Linux on Power

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

3.3.3 Cygwin on Windows 10

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

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

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

3.3.4 Mac OS

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

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

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

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

3.4 Controlling the number of threads

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

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

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

3.5 The number of MPI ranks per shared-memory unit

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

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

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

4 Overview of Functionality

WGSMP and *PWGSMP* are the primary routines for solving general sparse systems of linear equations and are described in detail in Sections 5 and 8, respectively. Additionally, the libraries contain some routines that provide a simpler interface to the solver (see Sections 6 and 9 for more details).

Both the serial/multithreaded and the message-passing parallel libraries allow the users to perform any appropriate subset of the following tasks: (1) Analysis and reordering, (2) LU factorization, (3) Back substitution, and (4) Iterative refinement. These functions can either be performed by calls to the primary serial and parallel subroutines *WGSMP* and *PWGSMP* (described in Sections 5 and 8, respectively), or by using the simpler serial and parallel interfaces (described in Sections 6 and 9, respectively). When using *WGSMP* or *PWGSMP* routines, *IPARM(2)* and *IPARM(3)* control the subset of the tasks to be performed. When using the simple interfaces, the tasks or the subsets of tasks to be performed are determined by the name of the routine.

WSMP and *PWSMP* libraries perform minimal input argument error-checking and it is the user's responsibility to call *WSMP* subroutines with correct arguments and valid options and matrices. In case of an invalid input, it is not uncommon for a routine to hang or to crash with segmentation fault. In the parallel version, on extremely rare occasions, insufficient memory can also cause a routine to hang or crash before all the processes/threads have had a chance to return safely with an error report. However, unlike the input argument and memory related errors, the numerical error checking capabilities of the computational routines are quite robust.

All *WSMP* routines can be called from Fortran as well as C or C++ programs using a single interface described in this document. As a matter of convention, symbols (function and variable names) are in capital letters in context of Fortran and in small letters in context of C. Please refer to Notes 5.2, 5.3, and 10.1 for more details on using *WSSMP* with Fortran or C programs.

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

4.1 Analysis and reordering

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

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

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

4.2 LU factorization

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

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

4.3 Back substitution

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

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

4.4 Iterative refinement

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

5 The Primary Serial/Multithreaded Subroutine: *WGSMP*

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

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

5.1 Types of matrices accepted and their input format

The *WGSMP* routine works for any non-singular square sparse matrix. Even if the original matrix is symmetric, *WGSMP* expects the entire sparse matrix as input. All floating point values must be 8-byte real numbers. All integers must be 4 bytes long unless you are using *libwsmp8_8.a*, which takes 8-byte integer inputs. Currently, two input formats are supported, namely, compressed sparse rows (CSR) and compressed sparse columns (CSC). Figure 1 illustrates both input formats; they are also explained briefly in Sections 5.3.2, 5.3.3, and 5.3.4.

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

5.2 Pivoting options

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

- **No pivoting:** Certain systems of linear equations do not require partial pivoting and the factorization is stable with the input sequence of rows. For such systems, *IPARM(8..12)* must be set to (0,0,0,0,0) to avoid the overhead of unnecessary pivoting. If the matrix is poorly scaled, then setting *IPARM(8..12)* to (0,1,0,0,0) will perform equilibration prior to factorization.

- **Threshold pivoting to control growth along columns:** For systems requiring partial pivoting, it is recommended that *IPARM(8..10)* be set to (0,1,1). Although these options do not directly affect pivoting, together they transform the matrix such that the magnitude of each diagonal entry is 1.0 and that of any nondiagonal entry is less than or equal to 1.0. Such transformation has the potential to significantly reduce the cost of pivoting during factorization. The recommended values of *IPARM(11..12)* are (1,0). The default values for *IPARM(8..12)* in *WGSMP* are (0,1,1,1,0). These are the same as the recommended values. The user may also experiment with (1,0,1,1,0) or (2,1,0,1,0) to see if a faster or more accurate factorization can be obtained by changing the default values in *IPARM(8..12)*.

  The pivoting threshold, which is a double precision value greater than 0.0 and less than or equal to 1.0, must be placed in *DPARM(11)*. The default value of pivoting threshold *DPARM(11)* is 0.01.

  Note 5.1 WGSMP *uses several mechanisms, other than partial pivoting, to enhance the accuracy of the final solution. These include a static permutation of rows maximize the diagonal product [2, 12, 15, 16], scaling, and iterative refinement in double and quadruple precision. Therefore, it is recommended that the smallest pivoting threshold that yields a solution with acceptable accuracy should be used. Minimizing row-interchanges associated with partial pivoting saves time and memory.*

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

$$
\begin{array}{c|ccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & 14. & & -5. & & & & -1. & -6. & \\
2 & & 14. & -1. & & -3. & & & & -1. \\
3 & -1. & & 16. & & & & -2. & & \\
4 & & & & 14. & & & & -3. & \\
5 & & & & & 14. & -1. & & & -1. \\
6 & & -2. & & -1. & & 16. & -2. & -4. & \\
7 & & & & -1. & & & 16. & & \\
8 & -3. & & -4. & & -3. & & -4. & 71. & \\
9 & & -1. & -2. & & & & & -4. & 16. \\
\hline
\end{array}
$$

A 9 X 9 general sparse matrix.

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

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

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

  Note that rook pivoting is not available in the message-passing parallel *PWGSMP* routine.

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

5.3 Calling sequence of the WGSMP subroutine

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

In the remainder of this document, the "system" refers to the sparse linear system of N equations of the form $AX = B$, where A is a general sparse coefficient matrix of dimension N, B is the right-hand-side vector/matrix and X is the solution vector/matrix, whose approximation $\overline{X}$ computed by *WGSMP* overwrites B when *WGSMP* is called to compute the solution of the system. The example program in *wgsmp_ex1.f* at the *WSMP* home page illustrates the use of the *WGSMP* subroutine for the matrix shown in Figure 1.

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

Note 5.3 *All user-callable* WSMP *and* PWSMP *routines expect their parameters to be passed by reference. Therefore, when calling* WGSMP *from a C program, the addresses of the parameters described in Section 5.3 must be passed.*

The calling sequence and description of the parameters of *WGSMP* is as follows. When an input data structure is not accessed in a particular call, a NULL pointer or any scalar can be passed as a place holder for that argument.

WGSMP (N, IA, JA, AVALS, B, LDB, NRHS, RMISC, IPARM, DPARM) void *wgsmp_(int *n, int ia[], int ja[], double avals[], double b[], double *ldb, int *nrhs, double rmisc[], int iparm[], int dparm[])*

5.3.1 N (type I): matrix dimension

```
INTEGER N
int *n
```

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

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

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

IA is an integer array of size one greater than N. *IA(I)* points to the first column (row) index of row (column) I in the array *JA* in CSR (CSC) format. Note that empty columns (or rows) are not permitted; i.e., IA($i + 1$) must be greater than IA(i).

Please refer to Figure 1 and description of *IPARM(4)* in Section 5.3.9 for more details.

5.3.3 JA (type I): column indices

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

The integer array *JA* contains the column (row) indices of the sparse matrix A stored in CSR (CSC) format. The column (row) indices of each row (column) must follow the indices of the previous column (row). Moreover, the column (row) indices should sorted in increasing order. *WSMP* provides two utility routines to sort the indices (see Section 10 for details).

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

```
DOUBLE PRECISION AVALS ( * )
double avals[]
```

The array *AVALS* contains the actual double precision values corresponding to the indices in *JA*. The size of *AVALS* is the same as that of *JA*. See Figure 1 for more details. Note that the analysis (ordering and symbolic factorization) phase of *WGSMP* accesses and uses *AVALS*—something that most conventional sparse solvers don't do.

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

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

The $N\times$ *NRHS* dense matrix B (stored in an *LDB* $\times$ *NRHS* array) contains the right-hand side of the system of equations $AX = B$ to be solved. If the number of right-hand side vectors, *NRHS*, is one, then B can simply be a vector of length N. During the solution, X overwrites B. If the solve (Task 3) and iterative refinement (Task 4) are performed separately, then the output of the solve phase is the input for iterative refinement. B is accessed only in the triangular solution and iterative refinement phases.

5.3.6 LDB (type I): leading dimension of B

```
INTEGER LDB
int *ldb
```

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

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

```
INTEGER NRHS
int *nrhs
```

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

5.3.8 RMISC (type O): double precision output info

```
DOUBLE PRECISION RMISC ( N, NRHS )
double rmisc[]
```

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

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

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

```
INTEGER IPARM ( 64 )
int iparm[64]
```

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

Note that all reserved entries; i.e., *IPARM(35:63)* must be filled with 0's on input.

- **IPARM(1) or iparm[0], type I or M:**

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

- **IPARM(2) or iparm[1], type M:**

  On input, *IPARM(2)* must contain the number of the starting task. On output, *IPARM(2)* contains 1 + number of the last task performed by *WGSMP*, if any. This is to facilitate users to restart processing on a problem from where the last call to *WGSMP* left it. Also, if *WGSMP* is called to perform multiple tasks in the same call and it

	IPARM			DPARM		
Index	Default	Description	Type	Default	Description	Type
1	mandatory I/P	default/user defined	M	-	elapsed time	O
2	mandatory I/P	starting task	M	-	first step	O
3	mandatory I/P	last task	I	-	unused	-
4	0	I/P format	I	-	largest pivot	O
5	1	numbering style	I	-	smallest pivot	O
6	3	max. # iter. refs.	M	2×10^{-15}	back err. lim.	I
7	3	residual norm type	I	-	backward error	O
8	0	max. matching use	I	-	unused	-
9	0	scaling w/o matching	I	-	unused	-
10	1	scaling w/ matching	I	10^{-18}	singularity threshold	I
11	1	thresh. pivoting opt.	I	0.01	pivot thresh.	I
12	0	pivot perturb. opt.	I	2×10^{-8}	small piv. thresh.	I
13	-	# row/col exchanges	O	-	# supernodes	O
14	-	# perturbations	O	-	# data-DAG edges	O
15	25	# factorizations	I	-	unused	-
16	1	ordering option 1	I	-	unused	-
17	0	ordering option 2	I	-	unused	-
18	0	ordering option 3	I	-	unused	-
19	0	ordering option 4	I	-	unused	-
20	0	ordering option 5	I	-	unused	-
21	1	block triangular form	I	-	structural symmetry	O
22	-	# blocks in B.T.F.	O	2×10^{-8}	small piv. repl.	I
23	-	actual $NNZ_L + NNZ_U$	O	-	actual fact. ops.	O
24	-	symbolic $NNZ_L + NNZ_U$	O	-	symbolic fact. ops.	O
25	0	RMISC use	I	5×10^{6}	min. parallel task size	I
26	-	# iter. ref. steps	O	1.0	supnode amalgamation	I
27	0	# fact. before re-analyze	I	1.0	re-analyze condition	I
$28^{S,T}$	0	rook pivoting	I	-	unused	-
29	0	garbage collection	I	-	unused	-
30	0	solve option	I	-	unused	-
31	1	# solves per factor	I	-	unused	-
32^{P}	0	block size	I	-	unused	-
33	-	no. of CPU's used	O	-	load imbalance	O
$34^{T,P}$	10	DAG manip. option	I	-	unused	-
35-63	0	reserved	R	0.0	reserved	R
64	-	return err. code	O	-	unused	-

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

returns with an error code in *IPARM(64)*, then the output in *IPARM(2)* indicates the task that failed. If *WGSMP* performs no task, then, on output, *IPARM(2)* is set to *max(IPARM(2),IPARM(3)*+ 1*)*. *WGSMP* can perform any set of consecutive tasks from the following list:

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

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

- **IPARM(3) or iparm[2], type I:**

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

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

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

- **IPARM(5) or iparm[4], type I:**

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

- **IPARM(6) or iparm[5], type I:**

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

  The default value of *IPARM(6)* is 3 for the unsymmetric solver.

- **IPARM(7) or iparm[6], type I:**

  If *IPARM(7)* = 0, 1, 2, or 3, then the residual in iterative refinement is computed in double precision (the same as the remainder of the computation). If *IPARM(7)* = 4, 5, 6, or 7, then the residual in iterative refinement is computed in quadruple precision (which is twice the precision of the remainder of the computation). If *IPARM(7)* = 0 or 4, then exactly *IPARM(6)* number of iterative refinement steps are performed without checking for the backward error. Additionally, in this case, if iterative refinement is not performed at all; i.e., if *IPARM(6)* = 0, then the residual is not calculated and returned in *DPARM(7)*. 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 backward error given by $\frac{\|b-A\overline{x}\|}{\|b\|}$ falls below the input value in *DPARM(6)*. Here A is the coefficient matrix, $\overline{x}$ is the computed solution, and b is the right-hand side vector. If *IPARM(7)* = 1 or 5, then 1-norms are used in computing the backward error, if *IPARM(7)* = 2 or 6, then 2-norms are used, and if *IPARM(7)* = 3 or 7, then infinity-norms are used. Moreover, if *IPARM(7)* = 1, 2, 3, 5, 6, or 7, then the actual backward error at the end of the last iterative refinement step is placed in *DPARM(7)*.

If *NRHS* > 1, then the maximum of the backward errors amongst the *NRHS* solution vectors is considered. Also note that, if scaling is performed (based on the inputs in *IPARM(9)* and *IPARM(10)*), then the backward errors are computed with respect to the scaled system and not the original system.

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

Note 5.5 *Please note that the residual is computed at the end of the solution step (Step 4). Even though* IPARM(7) *pertains to iterative refinement, it must be set to the appropriate value before the triangular solution step to be effective.*

Note 5.6 *Computing the residual adds a small overhead to the solution. Therefore, when solving a large number of linear systems w.r.t. the same factor without iterative refinement,* IPARM(7) *should be set to 0 to switch the residual computation off. This is important in applications in which the triangular solve time dominates.*

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

  WGSMP can use a maximum weight matching on the bipartite graph induced by the sparse coefficient matrix to permute its row such that the product of the absolute values of the diagonal is maximized [2, 12, 15, 16]. By default, indicated by *IPARM(8)* = 0, WSMP decides whether or not to use this matching depending on the structure and the values of coefficient matrix. If *IPARM(8)* is 1, then this permutation is always performed and if *IPARM(8)* is 2, then this permutation is not performed.

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

  During the analysis and reordering phase, depending on the input in *IPARM(8)*, *WGSMP* may use a maximum bipartite matching algorithm to permute the rows such that the product of the absolute values of the diagonal entries is maximized. In addition to a row permutation, the maximum matching algorithm also produces vectors for scaling the rows and the columns of the sparse matrix such that the magnitude of each diagonal entry of the scaled matrix is 1.0. If a maximum matching is not performed or if *IPARM(10)* is set to 0, then a simple equilibration can still be performed using *IPARM(9)*.

  Equilibration can be performed in multiple ways, and the desired equilibration method is communicated to *WGSMP* by the user via *IPARM(9)*.

  If *IPARM(9)* is set to -1, then the equilibration is not performed. If *IPARM(9)* is set to 0, then *WGSMP* automatically determines the best equilibration to apply. If *IPARM(9)* is set to 1, then row equilibration is performed. If *IPARM(9)* is set to 2, then row equilibration is followed by column equilibration. If *IPARM(9)* is set to 3, then column equilibration is performed. If *IPARM(9)* is set to 4, then column equilibration is followed by row equilibration.

  The default value of *IPARM(9)* is 0. Note that *IPARM(9)* is disregarded if a scaling based on maximum matching is performed.

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

  An input of *IPARM(10)* = 0 during numerical factorization implies that *WGSMP* will not perform a scaling of the input matrix using the vectors generated by applying the maximum bipartite matching algorithm to the input matrix. *IPARM(10)* = 1, which is the default, implies that such scaling is performed in an attempt to improve the numerical stability of factorization, if the row-permutation using the maximum bipartite matching is performed. If the maximum bipartite matching is not performed, then *IPARM(10)* is ignored. If *IPARM(9)* > 0 and *IPARM(10) = 1*, then *IPARM(10)* gets priority in determining how the scaling is performed.

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

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

  If *IPARM(11)* = 1 upon input, then threshold pivoting is performed using a pivoting threshold α ($0.0 < \alpha \leq 1.0$). The pivoting threshold α is equal to *DPARM(11)* if *DPARM(11)* > 0.0 on input (i.e., the user supplies the threshold). If *IPARM(11)* = 1 and *DPARM(11)* = 0.0, then *WGSMP* choses an appropriate threshold, which is placed in *DPARM(11)* as output. Threshold pivoting ensures that the pivot growth does not exceed $1/\alpha$ at any elimination step. Let d be the absolute value of the diagonal entry just before the i-th elimination step. Let r be the maximum absolute value among all entries in the i-th column. However, if $d < \alpha r$, then the i-th row can be exchanged by any row such that the absolute value of the entry in the i-th column of that row is greater than or equal to αr. If all entries in column i are zero (i.e., the matrix is singular), then the factorization is terminated and i is returned in *IPARM(64)*. A numbering from 1 to N is used to indicate this kind of failure, even if the input uses C-style numbering.

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

  Note 5.7 *The input in* IPARM(11) *must be set before the analyze phase so that it knows that partial pivoting is intended during numerical factorization. If the value in* IPARM(11) *is different during the analyze and factor phases, then the program may crash or generate incorrect results.*

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

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

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

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

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

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

  After factorization, *IPARM(13)* contains the number of diagonal entries that were perturbed in order to contain pivot growth. The perturbation, if any, is controlled by the user inputs in *IPARM(12)* and *DPARM(12)*.

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

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

0, then *WGSMP* assumes a very large number of factorizations per analysis step. If *IPARM(15)* = 1 or a small number, then *WGSMP* performs a faster analysis and reordering to minimize the overall run time, even though the factorization may run somewhat slower.

The default value of *IPARM(15)* is 25.

Note that if *IPARM(15)* is 1, then *WGSMP* assumes that another call to factor a matrix with the same structure will not be made before a call to the analysis step. Therefore, it may free some data structures after the factorization is complete, and an error or a crash may result if another factorization is attempted without performing the analysis step again.

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

  IPARM(16:20) control the ordering or the generation of the fill-reducing and load-balancing permutations for the input matrix.

  If *IPARM(16)* is -1, the ordering is not performed and the original ordering of columns is used. Note that the rows may still be permuted depending on the input in *IPARM(8)*. If *IPARM(16)* is -2, then reverse Cuthill-McKee ordering [4] is performed. If *IPARM(16)* is a nonnegative integer, then a graph-partitioning based ordering [8] is performed.

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

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

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

  WSMP uses graph-partitioning based ordering algorithms [10] 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) or iparm[17], type I:**

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

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

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

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

- **IPARM(20) or iparm[19], type I:**

  The input *IPARM(20)* lets the user communicate some known characteristics of the sparse matrix to *WGSMP* to aid it in choosing appropriate values of some internal parameters and to chose appropriate algorithms in various stages of ordering. If the user has no information about the type of sparse matrix or if the matrix does not fall into one of the categories below, then the default value 0 should be used.

  Certain sparse matrices have a very irregular structure and have a few rows/columns that are much denser than most of the rows/columns. For such matrices, the quality and the speed of ordering can usually be improved by setting *IPARM(20)* to 1.

  Sometimes, sparse matrices arise from finite-element graphs in which many or most vertices have more than one degree of freedom. In such graphs, there are a many small groups of nodes that share the same adjacency structure. If the sparse matrix comes from a problem like this, then a value of 2 should be used in *IPARM(20)*. This instructs *WGSMP* to construct a compressed graph before proceeding with the ordering, which then runs much faster as it runs on the smaller compressed graph rather than the original larger graph.

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

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

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

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

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

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

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

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

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

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

- **IPARM(25) or iparm[24], type I:**

  IPARM(25) = 0, which is the default, has no effect. If *IPARM(25)* = 1 during iterative refinement, then the component-wise backward error is returned in *RMISC*. If *IPARM(25)* = 1, then *RMISC* must point to a valid user-supplied double precision array of size N.

- **IPARM(26) or iparm[25], type O:**

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

- **IPARM(27) or iparm[26], type I:**

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

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

  The default value of *IPARM(28)* is 0, which results in the default pivoting method of *WSMP* that chooses a pivot row such that the diagonal element is not smaller in magnitude than the product of the pivoting threshold and the largest element in the pivot column. The magnitude of the diagonal element is not checked with respect to other elements in the pivot row. This method works for most sparse systems. However, in some cases, the resulting growth along the rows may yield unacceptable accuracy. If increasing the pivoting threshold does not bring the accuracy into an acceptable range, then the user may set *IPARM(28)* to 1. This ensures that the diagonal pivot is not smaller in magnitude than the pivoting threshold times the magnitude of any element in that row or column. This is known as rook pivoting and should be used only when absolutely necessary, because it has the potential to slow down the factorization considerably.

  Usually, in order to make rook pivoting effective, the user may have to suppress decomposition into a bock-triangular form by setting *IPARM(21)* to 0.

  Note that rook pivoting is not available in the message-passing parallel *PWGSMP* routine.

- **IPARM(29) or iparm[28], type I:**

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

  If an application requires several solve steps for each factorization step, then compaction of the factors resulting from garbage-collection may actually result in a slight increase in the performance of the solve steps and may be worthwhile.

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

  The default value of *IPARM(30)* is 0 and acceptable input values are 0, 1, 2, 4, 5, 6. If A is the coefficient matrix that is factored into lower-triangular L and upper-triangular U such that $A = LU$ and b is the right-hand side vector or matrix, then depending on the value of *IPARM(30)*, the following systems are solved (and B is overwritten by the solution x, as usual).

  - *IPARM(30)* = 0: $x = A^{-1}b$
  - *IPARM(30)* = 1: $x = L^{-1}b$
  - *IPARM(30)* = 2: $x = U^{-1}b$
  - *IPARM(30)* = 4: $x = (A^T)^{-1}b$
  - *IPARM(30)* = 5: $x = (U^T)^{-1}b$
  - *IPARM(30)* = 6: $x = (L^T)^{-1}b$

Note that two consecutive calls to *WGSMP* with *IPARM(1)* = *IPARM(2)* = 3, the first with *IPARM(30)* = 1 and the second with *IPARM(30)* = 2 is equivalent to a single call with *IPARM(1)* = *IPARM(2)* = 3 and *IPARM(30)* = 0. Also, using *IPARM(4)* = 1 and *IPARM(30)* = 4, 5, or 6 is mathematically equivalent to using *IPARM(4)* = 0 and *IPARM(30)* = 0, 1, or 2.

If the lower- and upper-triangular solves are performed separately using a value of *IPARM(30)* other than 0 or 4, then iterative refinement is switched off and backward error is not available as output. Another restriction with the use of separate lower- and upper-triangular solves is that they work correctly only when the option not to reduce the coefficient matrix to a block-triangular form is chosen by setting *IPARM(21)* = 0. This can cause the factorization time to increase.

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

  The user can set *IPARM(31)* (before the analysis step) to the expected number of triangular solve steps that would be performed for each factorization. This can help *WGSMP* in making some optimization decisions. By default, *IPARM(31)* is 1; i.e., it is assumed that each factorization will be followed by one call to the solve phase. Note that *IPARM(31)* is not the expected value of *NRHS* but the number of times the user expects to invoke the solution phase of *WGSMP* after each factorization step.

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

  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; otherwise, it uses the largest power of 2 less than or equal to *IPARM(32)*.

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

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

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

- **IPARM(34)^{T,P} or iparm[33], type I:**

  This parameter allows the user to affect the load-imbalance and communication and synchronization overhead versus fill-in trade-off to some extent. Depending on the number of CPUs being used, it attempts to manipulate the data-dependency graph to reduce load-imbalance and communication overhead at the cost of additional fill during factorization. *IPARM(34)*, whose default value is 10, controls the extent of reorganization of the data-dependency graph. Any integer value between 0 and $\log_2(P)$, where P is the number of CPUs, is a valid input. If *IPARM(34)* $> \log_2(P)$, then $\log_2(P)$ is used. The reorganization of the data-dependency graph can be completely switched off by setting *IPARM(34)* to 0.

- **IPARM(35:63) or iparm[34:62], type R:**

  These are reserved for future use.

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

  In the event of a successful return from *WGSMP* or *PWGSMP*, *IPARM(64)* is set to 0 on output. A nonzero value of *IPARM(64)* upon output is and error code and indicates that *WGSMP*/*PWGSMP* did not complete execution and detected an error condition. There are two types of error codes—negative and positive. In *PWGSMP*, the error code returned on all MPI processes is identical. The three least significant decimal digits indicate the error code and the remaining most significant digits indicate the MPI process number that was the first to encounter the error. For example, an error code of -700 indicates that process 0 detected error -700, and an error code of -2102 indicates that process 2 encountered error -102. The value of *IPARM(64)* will be set to -700 and -2102, respectively, upon return on all the processes.

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

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

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

An 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. Please refer to the description of *DPARM(25)* for ways of avoiding this error. The -200 error code is also returned if MPI is not initialized before a call to a *PWSMP* routine.

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

An output value of -700 for *IPARM(64)* indicates an internal error and should be reported to *wsmp@us.ibm.com*. Sometimes, error -700 is generated for very large matrices if the size of the factor exceeds 2^{31}, as a result of which, its indices cannot be stored using 4-byte integers. On some platforms, a special library *libwsmp8_8.a* is available. This library uses 8-byte integers and will solve the problem. Please make sure that all integer parameters that are passed to *WSMP* routines are of type *integer*8* (either declared explicitly, or by using the appropriate compiler option to promote all integers to 8-byte size) when using *libwsmp8_8.a*.

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

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

Note 5.8 *Note that in case of an out-of-memory error in the distributed-memory parallel solver, one or more of the input data arrays may be corrupted.*

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

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

The entries *DPARM(35)* through *DPARM(64)* are reserved. Unlike *IPARM*, only some of the first 34 entries of *DPARM* are used. The description of only the relevant entries of *DPARM* is given below. Note that all reserved entries, *DPARM(35:63)*, must contain 0.0.

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

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

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

  This output is set to -1.0 if nothing was done during the call, to 1.0 if *analysis* was the first step performed, to 2.0 if *factorization* was the first step performed, to 3.0 if *back substitution* was the first step performed, and to 4.0 if *iterative refinement* was the first step performed during the call.

In most applications, if the structure of the matrix stays unchanged and the values change only slightly from one factorization to the next, it is not necessary to repeat the analysis step. However, *WGSMP* and *PWGSMP* monitor the condition nnumber estimates and fill-in due to pivoting, and trigger a re-analysis if it is expected to improve the overall run time or accuracy. In this situation, *DPARM(2)* may return a 1.0 even if *IPARM(2)* was set to 2.

The inputs in *IPARM(2)* and *IPARM(3)* indicate the tasks that the user expects *WGSMP* or *PWGSMP* to perform. The ouputs in *IPARM(2)* and *DPARM(2)* indicate the tasks that were succesfully performed.

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

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

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

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

- **DPARM(6) or dparm[5], type I:**

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

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

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

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

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

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

  DPARM(11) is ignored if *IPARM(11)* = 0; else if *DPARM(11)* is $>$ 0.0, then it is used as the threshold for pivoting. The default value of *DPARM(11)* is 0.01; however, for most problems, the performance of the solver can be improved by reducing it to 0.001 or 0.0001 without any noticeable impact on accuracy.

  If *IPARM(11)* = 1 and *DPARM(11)* = 0.0 on input, then *WGSMP* chooses and appropriate pivoting threshold, and puts it in *DPARM(11)*. Please refer to the description of *IPARM(11)* for more details. *DPARM(11)* must be non-negative.

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

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

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

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

  Please see Note 5.9^P at the end of this section.

- **DPARM(14) or dparm[13], type O:**

  After the analysis step, *DPARM(14)* contains the number of edges in the data-dependency graph of the LU factorization. If *DPARM(14)* is less than or equal to *DPARM(13)* $-$ 1, then this graph is a tree or a forest of trees. A large value of *DPARM(14)* relative to *DPARM(13)* is indicative of higher potential overhead due to synchronization and data-copying.

  Please see Note 5.9^P at the end of this section.

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

  DPARM(21) returns the structural symmetry of the matrix (after various permutations of the original coefficient matrix) that is factored. This is a value between 0.0 and 1.0, where 1.0 indicates perfect structural symmetry and 0.0 indicates that there is no off-diagonal correspondence between the matrix and its transpose.

  Please see Note 5.9^P at the end of this section.

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

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

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

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

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

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

- **DPARM(25) or dparm[24], type I:**

  DPARM(25), whose default value is 5 million, is the minimum number of expected floating point operations in a task for it to be assigned to more than 1 CPU. This checks the granularity of parallelism and attempts to prevent the run time from increasing with the number of CPUs when the problem if not large enough to be effectively parallelized on the given number of CPUs. Although it is highly recommended that a value of 5 million or more be used, sometimes, for debugging or testing purposes, a user may want to solve a very small problem in parallel. In such situations, a smaller value of *DPARM(25)* (minimum 1) can be used to avoid error code -200 (see *IPARM(64)*). However, for obtaining the best performance, *DPARM(25)* must be set to the default or a higher value.

- **DPARM(26) or dparm[25], type I:**

  This input controls the degree of supernode amalgamation performed by *WGSMP*. In addition to groups of rows-column pairs with the same nonzero structure in the LU factors, *WGSMP* often combines consecutive rows and columns whose nonzero structure closely matches but is not identical. This is done to enhance the efficiency and parallelism of the BLAS routines by increasing the sizes of the supernodes that these routines are called to work on. The default level of aggressiveness with which such supernode amalgamation is performed in *WGSMP* corresponds to the default value of 1.0 for *DPARM(26)*. Increasing *DPARM(26)* will increase the number of floating-point operations, but may also increase the factorization Megaflops rate. Reducing *DPARM(26)* below 1.0 will have the opposite effect. The user may experiment with nearby values, such as 1.2 and 0.8 to see if increasing or decreasing the degree of amalgamation improves the overall performance on the application at hand.

- **DPARM(27) or dparm[26], type I:**

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

- **DPARM(33)^P or dparm[32], type O:**

  At the end of the analysis phase, the output in *DPARM(33)* gives a rough indication of the fraction of the execution time that is expected to be the load-imbalance overhead ($0.0 \leq$ *DPARM(33)* < 1.0).

Note 5.9 *Some* IPARM *and* DPARM *outputs in the message-passing parallel version of the software are fragile and are valid only if the number of block-triangular blocks is 1; i.e., either* IPARM(21) *is 0 on input or* IPARM(22) *is 1 on output. Currently,* DPARM(13), DPARM(14), DPARM(21), *and* DPARM(33) *are fragile outputs.*

6 Subroutines Providing a Simpler Serial/Multithreaded Interface

In this section, we describe a simpler interface to *WGSMP*. This interface accepts the input in both CSR and CSC formats and expects a Fortran-style indexing starting from 1. The shape, size, attributes, and meaning of all data structures is the same as in the calling sequence of the *WGSMP* routine described in Section 5. The *WSMP* home page contains an example driver program *wgsmp_ex2.f* that uses the simple interface.

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

WGCALZ (N, IA, JA, AVALS, NNZ, OPC, INFO)
void *wgcalz_ (int *n, int *ia, int *ja, double *avals, int *nnz, double *opc, int *info)*

WGRALZ (N, IA, JA, AVALS, NNZ, OPC, INFO)
void *wgralz_ (int *n, int *ia, int *ja, double *avals, int *nnz, double *opc, int *info)*

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

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

WGCLUF (N, IA, JA, AVALS, THRESH, INFO)
void *wgcluf_ (int *n, int *ia, int *ja, double *avals, double *thresh, int *info)*

WGRLUF (N, IA, JA, AVALS, THRESH, INFO)
void *wgrluf_ (int *n, int *ia, int *ja, double *avals, double *thresh, int *info)*

These routines perform LU factorization. The descriptions of N, *IA*, *JA*, and *AVALS* are the same as in Section 5.3. *THRESH* is a double precision input that must contain the pivoting threshold, a double precision value between 0.0 and 1.0 (both inclusive). If *THRESH* is 0.0, then partial pivoting is not performed. A value of 0.01 is recommended and yields fast and accurate results for most sparse systems. *INFO* is an integer output that is identical to *IPARM(64)* as described in Section 5.3.9.

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

WGCSLV (N, IA, JA, AVALS, B, LDB, NRHS, NITER, BERR, INFO)
void *wgcslv_ (int *n, int *ia, int *ja, double *avals, double *b, int *ldb, int *nrhs, int *niter, double *berr, int *info)*

WGRSLV (N, IA, JA, AVALS, B, LDB, NRHS, NITER, BERR, INFO)
void *wgrslv_ (int *n, int *ia, int *ja, double *avals, double *b, int *ldb, int *nrhs, int *niter, double *berr, int *info)*

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

7 Replacing Rows or Columns and Updating Triangular Factors^{S,T}

This section is relevant only for the serial/multithreaded library. The functions described in this section are not implemented in the message-passing library. Just like other *WSMP* routines, these can be called from a C program by passing the arguments by reference (Note 5.3).

In this section, we discuss how *WSMP*'s general sparse solver can be used to update an LU factorization. This has applications in some Operations Research algorithms, particularly the Simplex algorithm. We use the well-known Forrest-Tomlin [3] method to implement the row or column updates in *WSMP*. If *WSMP* is used in an application that requires updating the factors, then only the routines described in this section should be used. These routines allow the user to perform the analysis (*WU_ANALYZ*) and LU factorization (*WU_FACTOR*) steps on an $n \times n$ sparse matrix A, compute $A^{-1}b$ (*WU_FTRAN*) and $(A^T)^{-1}b$ (*WU_BTRAN*), and update the factors (*WU_UPDATE*) such that the new factors represent the factorization of a matrix A' in which all columns are the same as in A except column q, which is replaced by a sparse vector a_q. All triangular solves using *WU_FTRAN* and *WU_BTRAN* routines are performed with respect to the last updated factors. After several updates, it may be necessary to refactor the new *basis* (current version of the coefficient matrix after the updates) because the speed and the accuracy of the triangular solves may decline slightly with each update. However, in addition to speed and accuracy considerations, *WSMP* imposes a hard limit of n on the maximum number of updates that can be performed before a refactorization with calls to *WU_ANALYZ* and *WU_FACTOR* routines is necessary. The output *INFO* will return -700 if more than n updates are attempted without refactoring the basis.

Note 7.1 *Note that the routines in this section are geared only towards replacing columns of the sparse input matrix A. However, row-replacement can easily be emulated by the reversing the* RC *parameter in* WU_ANALYZ *and reversing the roles of* WU_FTRAN *and* WU_BTRAN.

Note 7.2 *At the present time, the factor-updating facility described in this section is planned only for the serial and shared-memory parallel version of* WSMP.

We now describe the routines and their calling sequences in greater detail.

7.1 WU_ANALYZ (analysis)

WU_ANALYZ (RC, NUMB, N, IA, JA, AVALS, INFO)

WU_ANALYZ performs fill-reducing ordering and symbolic factorization.

- RC (integer input): If the input matrix is in the compressed sparse row (CSR) format, then *RC* should be 0 and if the input matrix is in the compressed sparse column (CSC) format, then *RC* should be 1.
- NUMB (integer input): *NUMB* = 0 indicates C-style numbering and indexing convention starting from 0 and *NUMB* = 1 indicates the Fortran convention of indices starting from 1 in *IA* and *JA*.
- N (integer input): N is the number of rows and columns in the matrix.
- IA (integer input array of size N): See Section 5.3.2.
- JA (integer input array of size IA(N+NUMB)−NUMB): See Section 5.3.3.
- AVALS (double precision array of size IA(N+NUMB)−NUMB): See Section 5.3.4.
- INFO (integer output): Identical to *IPARM(64)* described in Section 5.3.9.

7.2 WU_FACTOR (factor)

WU_FACTOR (IA, JA, AVALS, THRESH, RCOND, INFO)

WU_FACTOR factors the $N \times N$ sparse basis stored in *IA*, *JA*, and *AVALS*, where N is the same as in the most recent call to *WU_ANALYZ*. The numbering and format of *IA*, *JA*, and *AVALS* should also be the same as in *WU_ANALYZ* that precedes the call to this routine. All parameters have the same description as in *WU_ANALYZ* except *THRESH* and *RCOND*. *THRESH* is a double precision input and determines the pivoting threshold to be used during LU factorization. Valid values of *THRESH* are from 0.0 to 1.0, both inclusive. We recommend using a value around 0.1 as the *THRESH* input. *RCOND* is a double precision output that contains the inverse of a crude condition number estimate of the coefficient matrix. A very small value of *RCOND* implies a large condition number, which may suggest that the solutions from the factorization may contain large errors.

7.3 WU_UPDATE (update)

WU_UPDATE (Q, NZQ, AQINDX, AQVALS, ACC, INFO)

WU_UPDATE updates a previously performed factorization by replacing the original column Q by the new sparse column stored in *AQINDX* and *AQVALS*.

- Q (integer input): Q is the column number that is to be replaced. The range of valid values for Q is *NUMB* to $N - 1 +$ *NUMB*, where N and *NUMB* are the same that were used in the most recent call to *WU_ANALYZ*.
- NZQ (integer input): *NZQ* is the number of nonzeros in the new column Q that replaces the existing column Q.
- AQINDX (integer input array of size NZQ): *AQINDX* contains the indices of all the rows that have a nonzero in the new column Q. The indices must be sorted in increasing order.

- AQVALS (double precision input array of size NZQ): *AQVALS* contains the values in the incoming column Q corresponding to the row-indices in *AQINDX*.
- ACC (double precision input and output): Although Forrest-Tomlin update does not perform numerical pivoting, it has been shown to be quite accurate in practice. As explained in [3] (Equation 4.1), the Forrest-Tomlin method also provides a mechanism to check the accuracy of the update so that the user can refactor the basis if the accuracy falls below (or the double precision output *ACC* rises above) a certain threshold. The accuracy check involves comparing two quantities $\overline{\alpha}_q^q$ and $\alpha_q u_{q,q}$ described in [3] (q is the index of the column that was last updated). The routine *WU_UPDATE* returns the following double precision value in *ACC* if and only if *ACC* is set to 0.0 on input:

  $$ACC = \frac{|\overline{\alpha}_q^q - \alpha_q u_{q,q}|}{\max(|\overline{\alpha}_q^q|, |\alpha_q u_{q,q}|)}$$

  The accuracy check increases the update time by about 50%. Therefore, it is not performed if *ACC* has a nonzero input value, in which case, *ACC* remains unchanged.

  The subroutine *WU_RESID* (Section 7.10) can be used as an alternative to using *ACC* for determining the accuracy of the solution.
- INFO (integer output): Identical to *IPARM(64)* described in Section 5.3.9.

7.4 WU_FTRAN (forward solve)

WU_FTRAN (B, LDB, X, LDX, NRHS, INFO)

WU_FTRAN computes $x = A^{-1}b$, where the $N\times$ *NRHS* matrix b is stored in the *LDB* $\times$ *NRHS* input array B and the $N\times$ *NRHS* matrix x is stored in the *LDX* $\times$ *NRHS* output array X. *LDB*, the leading dimension of B and *LDX*, the leading dimension of X, must be greater than or equal to N. Any error condition encountered is reported in the output *INFO*, whose description is the same as that of *IPARM(64)* in Section 5.3.9.

7.5 WU_BTRAN (backward solve)

WU_BTRAN (B, LDB, X, LDX, NRHS, INFO)

WU_BTRAN computes $x = (A^T)^{-1}b$. Its calling sequence and parameter description is same as that of *WU_FTRAN*.

7.6 WU_UPDFTR (update followed by forward solve)

WU_UPDFTR (Q, NZQ, AQINDX, AQVALS, ACC, B, LDB, X, LDX, NRHS, INFO)

WU_UPDFTR is semantically equivalent to a call to *WU_UPDATE* immediately followed by a call to *WU_FTRAN*; however it is faster than making the two calls separately. The description of all parameters is the same as in Sections 7.3 and 7.4.

7.7 WU_UPDBTR (update followed by backward solve)

WU_UPDBTR (Q, NZQ, AQINDX, AQVALS, ACC, B, LDB, X, LDX, NRHS, INFO)

WU_UPDBTR is semantically equivalent to a call to *WU_UPDATE* immediately followed by a call to *WU_BTRAN*; however it is faster than making the two calls separately. The description of all parameters is the same as in Sections 7.3 and 7.5.

7.8 WU_FTRUPD (forward solve followed by update)

WU_FTRUPD (B, LDB, X, LDX, NRHS, Q, NZQ, AQINDX, AQVALS, ACC, INFO)

WU_FTRUPD is semantically equivalent to a call to *WU_UPDATE* immediately following a call to *WU_FTRAN*; however it is faster than making the two calls separately. The description of all parameters is the same as in Sections 7.3 and 7.4.

7.9 WU_BTRUPD (backward solve followed by update)

WU_BTRUPD (B, LDB, X, LDX, NRHS, Q, NZQ, AQINDX, AQVALS, ACC, INFO)

WU_BTRUPD is semantically equivalent to a call to *WU_UPDATE* immediately following a call to *WU_BTRAN*; however it is faster than making the two calls separately. The description of all parameters is the same as in Sections 7.3 and 7.5.

7.10 WU_RESID (compute backward error)

WU_RESID (B, LDB, X, LDX, NRHS, RESID, LDR, BERR, FERR, INFO)

WU_RESID computes the residual $Ax - b$ or $A^T x - b$ and reports it in the *LDR* $\times$ *NRHS* double precision output array *RESID*. It also computes the sparse backward error $\frac{\|b-Ax\|}{\|b\|}$ and reports it in the output double precision scalar *BERR*. The latest updated version of the matrix A is used. The *LDB* $\times$ *NRHS* input array B contains the right-hand side matrix b and the *LDX* $\times$ *NRHS* input array X contains the solution matrix x, which must have been computed by an earlier call to *WU_FTRAN*, *WU_BTRAN*, *WU_UPDFTR*, or *WU_UPDBTR*. The residual and the backward error cannot be computed after a solution obtained by routines *WU_FTRUPD* and *WU_BTRUPD*. The residual obtained in *RESID* can be used to implement iterative refinement [5].

The double precision output *FERR* contains an estimation of the forward error; i.e., the distance of the obtained solution from the actual solution. *FERR* computation is slightly expensive, and therefore, is performed only if *FERR* is set to 0.0 on input. If *FERR* is not 0.0 on input, then it is returned unchanged and forward error estimation is not performed. *FERR* computation, if desired, needs to performed only once after a factorization or an update. Unlike *BERR*, *FERR* is independent of B.

Besides iterative refinement, this routine can also be used as an alternative to computing *ACC* output of the update routines for accuracy determination. The user, however, must distinguish between the interpretation of *BERR*, *FERR* and *ACC*. *ACC* determines the accuracy of the update only. For an accurate update of an ill-conditioned matrix, *ACC* will be small, but *BERR*, and particularly *FERR*, could be large. Therefore, large values of *BERR* and *FERR* alone should not be used to judge the accuracy of the updates and hence for determining whether or not to refactor the basis. An increase in *BERR* over the previous iteration may point to a loss of accuracy in the update.

7.11 WU_BSIZE (size of current basis)

WU_BSIZE (BSIZE, NUPDATES)

This routine returns the number of nonzeros in the basis after the most recent update in the integer output parameter *BSIZE*. It returns the number of updates performed since the last factorization in the integer output *NUPDATES*. This routine is useful if the user wishes to use *WS_BASIS* (Section 7.12) to return the current basis for possible refactorization.

7.12 WU_BASIS (return current basis)

WU_BASIS (IA, JA, AVALS, INFO)

WU_BASIS returns the current basis in the same format in which the original basis was made available to *WU_ANALYZ* and *WU_FACTOR* earlier (i.e., same values of *RC* and *NUMB* apply). All parameters in *WU_BASIS* are output parameters. The user must provide an integer array of size $N+1$ in *IA*, an integer array of size *BSIZE* in *JA* and a double precision array of size *BSIZE* in *AVALS*, where N is the dimension of the basis that was used in the most recent call to *WU_ANALYZ* and *BSIZE* is the number of nonzeros in the basis obtained by a call to *WS_BASIS* (Section 7.11).

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

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

Note that *PWGSMP* requires a thread-safe version of MPI if using more than one thread per process. As a result, when used with MPICH, the number of computational threads should be set to 1. Please refer to Section 3.4 for details on controlling the number of threads usd by *WSMP*.

8.1 Parallel data-distribution

If the parallel program is running on p MPI processes, we shall name the processes $P_0, P_1, \ldots, P_{p-1}$. In general, P_i initially owns N_i rows (CSR) or columns (CSC) of the coefficient matrix A and N_i rows of the right-hand side B. The dimension of the system of equations is $N = \Sigma_{i=0}^{p-1} N_i$. There is no restriction on the relative amount of data on any of the processes; the permitted values of all N_is are from 0 to N. Figure 2 illustrates the input data structures for the matrix A for $p = 3$, $N_0 = 3$, $N_1 = 3$, and $N_2 = 3$ for the matrix shown in Figure 1 earlier. Note that it is not necessary for all processes to start with the same number of rows or columns of the matrix. However, consecutive processes must contain consecutive portions of the matrix A (and also the right-hand side B). In other words, if l is the last row/column on process P_i, then the first row/column on process P_{i+1} must be $l+1$. In addition, the indices and the values corresponding to consecutive rows/columns must appear in consecutive order, just as in the serial/multithreaded version.

8.2 Calling sequence

The message-passing parallel routine *PWGSMP* must be called on all the processes. The calling sequence on process P_i is as follows ($0 \leq i < p$):

PWGSMP (N_i, IA_i, JA_i, $AVALS_i$, B_i, LDB_i, NRHS, $RMSIC_i$, IPARM, DPARM)

void *pwgsmp_(int *n_i, int ia_i[], int ja_i[], double $avals_i$[], double b_i[], int *ldb_i, int *nrhs, double $rmisc_i$[], int iparm[], double dparm[])*

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

Following is a brief description of the arguments. A more detailed description can be found in Section 5.3; this section is intended to highlight the differences between the serial/multithreaded and the message-passing versions, wherever applicable.

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

©IBM Corporation 1997, 2021. All Rights Reserved.

		CSC Format			CSR Format		
Node#	K	IA(K)	JA(K)	AVALS(K)	IA(K)	JA(K)	AVALS(K)
P_0	1	1	1	14.0	1	1	14.0
	2	4	3	-1.0	5	3	-5.0
	3	7	8	-3.0	9	7	-1.0
	4	12	2	14.0	12	8	-6.0
	5		6	-2.0		2	14.0
	6		9	-1.0		3	-1.0
	7		1	-5.0		5	-3.0
	8		2	-1.0		9	-1.0
	9		3	16.0		1	-1.0
	10		8	-4.0		3	16.0
	11		9	-2.0		7	-2.0
P_1	1	1	4	14.0	1	4	14.0
	2	4	6	-1.0	3	8	-3.0
	3	7	7	-1.0	6	5	14.0
	4	9	2	-3.0	11	6	-1.0
	5		5	14.0		9	-1.0
	6		8	-3.0		2	-2.0
	7		5	-1.0		4	-1.0
	8		6	16.0		6	16.0
	9					7	-2.0
	10					8	-4.0
P_2	1	1	1	-1.0	1	4	-1.0
	2	6	3	-2.0	3	7	16.0
	3	11	6	-2.0	8	1	-3.0
	4	14	7	16.0	12	3	-4.0
	5		8	-4.0		5	-3.0
	6		1	-6.0		7	-4.0
	7		4	-3.0		8	71.0
	8		6	-4.0		2	-1.0
	9		8	71.0		3	-2.0
	10		9	-4.0		8	-4.0
	11		2	-1.0		9	16.0
	12		5	-1.0			
	13		9	16.0			

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

A 9 X 9 general sparse matrix.

The storage of this matrix in the input formats accepted by PWGSMP on 3 processes is shown in the table.

Figure 2: A sample distribution of the coefficient matrix in two input formats for the distributed-memory parallel *PWGSMP* routines on three processes.

Note that, the distribution chosen for a given matrix, cannot be changed between different phases of the solution process. 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.

- **IA$_i$:** Integer array of size $N_i + 1$. This array provides pointers into the array of indices *JA*. See Figure 2 for more details. Note that if N_i = 0, then *IA$_i$* must be a single integer with a value of 0 (with C-style numbering) or 1 (with Fortran-style numbering) to be consistent with the definition of *IA$_i$*.

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

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

- **B$_i$, LDB$_i$, *and* NRHS:** B is a double precision array of size *LDB$_i$*$\times$ *NRHS*, where *LDB$_i$* $\geq N_i$. If N_i = 0, then B can be a *NULL* pointer. The number of right-hand sides, *NRHS*, must be the same on all processes.

- **RMISC$_i$:** Double precision array of size *LDB$_i$*$\times$ *NRHS*. The output contains component-wise backward error corresponding to B_i.

- **IPARM *and* DPARM:** The description of *IPARM* and *DPARM* is contained in Sections 5.3.9 and 5.3.10, respectively. For the message-passing parallel version, all input parameters in these arrays must be identical on each process.

  In the message-passing parallel version, *IPARM(28)* is ignored. *IPARM(32)* is used only in the message-passing version. Also, please refer to Note 5.9.

9 Parallel Subroutines Providing a Simpler Interface

In this section, we list the routines that provide a simpler interface to *PWGSMP*. This interface is analogous to the simple interface to *WGSMP* described in Section 6. Parallel routines *PWGxALZ, PWGxLUF,* and *PWGSLV* are available to the users, where x is C for the CSC input format and R for the CSR input format. The function and the calling sequence of these routines are identical to the serial/multithreaded routines *WGxALZ, WGxLUF,* and *WGSLV*, respectively described in Section 6. The meaning of the various parameters is the same as in the calling sequence of the *PWGSMP* routine described in Section 8.

Note 9.1 *The calls to the* WGSMP/PWGSMP *routines should not be mixed with those to the routines in the simple interface described here and in Section 6. The user must choose to use either the* WGSMP/PWGSMP *routines or the simple interface for a given application, and stick to the chosen interface.*

10 Miscellaneous Routines

In this section, we describe some optional routines available to the users for managing memory allocation, data distribution, and some other miscellaneous tasks. Just like other *WSMP* routines, these can be called from a C program by passing the arguments by reference (Note 5.3).

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

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

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

Please read Note 10.1 at the beginning of this section.

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

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

Please read Note 10.1 at the beginning of this section.

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

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

Please read Note 10.1 at the beginning of this section.

10.4 *WSETMAXTHRDS (NUMTHRDS)*

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

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

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

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

10.5 *WSSYSTEMSCOPE* and *WSPROCESSSCOPE*

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

10.6 *WSETMAXSTACK (FSTK)*

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

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

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

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

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

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

10.8 *WSETNOBIGMAL ()*

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

10.9 *WSMP_VERSION (V, R, M)*

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

Please read Note 10.1 at the beginning of this section.

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

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

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

Please read Note 10.1 at the beginning of this section.

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

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

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

Please read Note 10.1 at the beginning of this section.

10.12 *WGFFREE ()*^{S,T} and *PWGFFREE ()*^{P}

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

10.13 *WGSFREE ()*^{S,T} and *PWGSFREE ()*^{P}

The routines *WGFFREE* and *PWGFFREE* described in Sections 10.12 release the memory occupied by the factors of the coefficient matrix, but retain all other data-structures to facilitate subsequent factorizations of matrices of the same size and nonzero pattern. *WGSFREE* and *PWGSFREE* release *all* the memory allocated by *WSMP* in the context of solving unsymmetric systems via direct factorization. If you need to solve more unsymmetric systems after call to *WGSFREE* or *PWGSFREE*, then you must start with the analysis step.

10.14 *WGSMATVEC (N, IA, JA, AVALS, X, B, FMT, IERR)*^{S}

This routine multiplies the vector X with the N-dimensional general sparse matrix stored in *IA, JA, AVALS* and returns the result in the vector B. The description of *N, IA, JA,* and *AVALS* is the same as in Section 5.3. *FMT* is an integer input; an input value of 1 is used to indicate that the matrix is stored in the CSR format and an input value of 2 is used to indicate that the matrix is stored in the CSC format. *IERR* is equivalent to *IPARM(64)*, described in Section 8.2.9. Both C and Fortran style numbering convention is supported.

Note that this routine is neither multithreaded, nor optimized for performance. A multithreaded and optimized version of sparse matrix vector multiplication is a part of the recently released iterative solver package [9].

10.15 *PWGSMATVEC (*N_i*,* IA_i*,* JA_i*,* $AVALS_i$*,* X_i*,* B_i*, FMT, IERR)*^{P}

This routine multiplies the vector X with the general sparse matrix stored in *IA, JA, AVALS* and returns the result in the vector B. Here N_i is the local number of rows/columns on Process i and the local number of entries of the distributed vectors X and B stored on it. The matrix as well as both the vectors are expected to be stored in a distributed fashion, similar to the distribution illustrated in Figure 2. The description of N_i, IA_i, JA_i, and $AVALS_i$ is the same as in Section 8.2. *FMT* is an integer input; an input value of 1 is used to indicate that the matrix is stored in the CSR format and an input value of 2 is used to indicate that the matrix is stored in the CSC format. *IERR* is equivalent to *IPARM(64)*, described in Section 8.2.9. Both C and Fortran style numbering convention is supported.

10.16 *WSETMPICOMM (INPCOMM)*^{P}

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

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

Note 10.2 INPCOMM *must be a communicator generated by MPI's Fortran interface. If you are using the* PWSMP *library from a C/C++ program and using a communicator other than* MPI_COMM_WORLD*, then you would need to use* MPI_Comm_c2f *to obtain the equivalent Fortran communicator, or write a small Fortran routine that would generate a communicator over the same processes as your C communicator.*

11 Routines for Double Complex Data Type

The double complex (complex*16) version of the unsymmetric/general solver can be accessed via routines *ZGSMP, ZGRALZ, ZGCALZ, ZGRLUF, ZGCLUF, ZGRSLV, ZGCSLV,* and *ZGSMATVEC* for the serial/multithreaded version and *PZGSMP, PZGRALZ, PZGCALZ, PZGRLUF, PZGCLUF, PZGRSLV, PZGCSLV,* and *PZGSMATVEC* for the message-passing version. These routines are identical to their double precision real counterparts described in Sections 5 and 6, except that the data type of *AVALS*, *B*, and *RMISC* in these routines is *double complex* or *complex*16*. The *WSMP* web page at *http://www.research.ibm.com/projects/wsmp* has an example program *zgsmp_ex1.f* that solves a system of linear equations with complex coefficients and solution and RHS vectors.

Note that it is wasteful to use the unsymmetric solver for Hermitian matrices. The symmetric solver is equipped to handle these (please refer to the documentation for the symmetric solver).

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

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

13 Acknowledgements

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

References

[1] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report TR-93-018, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1993.

[2] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[3] John J. Forrest and John A. Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming*, 2:263–278, 1972.

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

[5] Gene H. Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1996.

[6] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 24(2):529–552, 2002.

[7] Anshul Gupta. A shared- and distributed-memory parallel general sparse direct solver. *Applicable Algebra in Engineering, Communication, and Computing*, 18(3):263–277, 2007.

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

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

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

[11] Anshul Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3):301–324, September 2002.

[12] Anshul Gupta and Lexing Ying. On algorithms for finding maximum matchings in bipartite graphs. Technical Report RC 21576, IBM T. J. Watson Research Center, Yorktown Heights, NY, October 1999.

[13] Steven M. Hadfield. *On the LU Factorization of Sequences of Identically Structured Sparse Matrices within a Distributed Memory Environment*. PhD thesis, University of Florida, Gainsville, FL, 1994.

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

[15] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *SC98 Proceedings*, 1998.

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