RC24398 (W0711-017) November 5, 2007
Computer Science

IBM Research Report

WSMP: Watson Sparse Matrix Package
Part III: Iterative Solution of Sparse Systems
Version 7.11

(http://www.alphaworks.ibm.com/tech/wsmp)

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

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

© IBM Corporation 1997, 2007. All Rights Reserved.

Contents

1 Introduction to Part III **4**

2 Obtaining, Linking, and Running *WSMP* **4**
2.1 AIX on IBM pSeries platform . . . 5
2.1.1 Linking the libraries . . . 5
2.1.2 Setting environment variables and running . . . 6
2.1.3 Linking and execution-time problems . . . 6
2.2 Linux on IA32 platform . . . 6
2.2.1 Linking the libraries . . . 7
2.2.2 Setting environment variables and running . . . 7
2.2.3 Linking and execution-time problems . . . 7
2.3 Linux on IA64 and x86_64 platforms . . . 7
2.4 Linux on AMD processors . . . 8
2.5 SGI Altix on IA64 processors . . . 8
2.6 Other platforms . . . 8
2.6.1 SunOS on UltraSPARC or Opteron . . . 8
2.6.2 Tru64 Unix on Compaq(HP) Alpha processors . . . 9

3 Description of Functionality **9**
3.1 Types of matrices accepted and their input format . . . 11
3.2 Krylov subspace solvers supported . . . 11
3.3 Preconditioners and their parameters . . . 11
3.4 Calling sequence of the WISMP subroutine . . . 11
3.4.1 N (type I): matrix dimension . . . 13
3.4.2 IA (type I): row/column pointers . . . 13
3.4.3 JA (type I or M): column/row indices . . . 13
3.4.4 AVALS (type I or M): nonzero values of the coefficient matrix . . . 13
3.4.5 B (type I or O): right-hand side vector/matrix . . . 14
3.4.6 LDB (type I): leading dimension of B . . . 14
3.4.7 X (type O or I): solution vector/matrix . . . 14
3.4.8 LDX (type I): leading dimension of X . . . 14
3.4.9 NRHS (type I): number of right-hand sides . . . 14
3.4.10 RMISC (type I, O, M): double precision output info . . . 14
3.4.11 CVGH (type O): double precision convergence history output . . . 15
3.4.12 IPARM (type I, O, M, and R): integer array of parameters . . . 15
3.4.13 DPARM (type I, O, M, and R): double precision parameter array . . . 25

4 Miscellaneous Routines **27**
4.1 *WS_SORTINDICES_I (M, N, IA, JA, INFO)* ^{S,T} . . . 27
4.2 *WS_SORTINDICES_D (M, N, IA, JA, AVALS, INFO)* ^{S,T} . . . 27
4.3 *WS_SORTINDICES_Z (M, N, IA, JA, AVALS, INFO)* ^{S,T} . . . 27
4.4 *WSETMAXTHRDS (MAXTHRD)* . . . 28
4.5 *WSSYSTEMSCOPE* and *WSPROCESSSCOPE* . . . 28
4.6 *WSETMAXSTACK (FSTK)* . . . 28
4.7 *WSETLF (DLF)* ^{T,P} . . . 28
4.8 *WSETNOBIGMAL ()* . . . 29
4.9 *WSMP_VERSION (V, R, M)* . . . 29
4.10 *WSMP_INITIALIZE ()* ^{S,T} and *PWSMP_INITIALIZE ()* ^{P} . . . 29
4.11 *WSMP_CLEAR ()* ^{S,T} and *PWSMP_CLEAR ()* ^{P} . . . 29

4.12 *WISFREE ()*^{S,T} and *PWISFREE ()*^{P} 29

5 Support for Double Complex Data Type 30

6 Notice: Terms and Conditions for Use of *WSMP* and *PWSMP* 30

7 Acknowledgements 30

1 Introduction to Part III

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 using a direct method. It can be used as a serial package, or in a shared-memory multiprocessor environment, or as a scalable parallel solver in a message-passing environment, where each node can either be a uniprocessor or a shared-memory multiprocessor. *WSMP* is comprised of three parts. Part I uses direct factorization for solving symmetric systems without numerical pivoting. Part II sparse LU factorization with pivoting for numerical stability to solve general systems. This document describes the iterative solution of sparse systems of linear equations in *WSMP*. Parts I and II of User's Guide [6] can be obtained from *http://www.cs.umn.edu/˜agupta/wsmp.html*, 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.

Note 1.1 *The current version supports iterative solution of linear systems only in the serial/multithreaded mode. The distributed-memory parallel solvers will be made available shortly. Please disregard any portions of this document that refer to distributed-memory/message-passing parallelism.*

Note 1.2 *Although* WSMP *and* PWSMP *libraries contain multithreaded code, the libraries themselves are* **not** *thread-safe. Therefore, the calling program cannot invoke multiple instances of the routines contained in* WSMP *and* PWSMP *libraries from different threads at the same time.*

The organization of this document is as follows. Section 2 lists the various libraries that are available and describe how to obtain and use the libraries. Section 3 describes the functionality of the main serial/multithreaded routine that provides an advanced single-routine interface to the entire software. This section also describes the input data structures for the serial and multithreaded cases. Section 4 describes a few utility routines available to the users. Section 5 gives a brief description of the double-complex data type interface of *WSMP*. Section 6 contains the terms and conditions that all users of the package must adhere to.

2 Obtaining, Linking, and Running *WSMP*

The software can either be downloaded from IBM's *alphaWorks* site at *http://www.alphaworks.ibm.com/tech/wsmp*. The main directory of your platform contains a file *wsmp.lic*. This file must always be present in the directory from which you are running a program linked with 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.

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@watson.ibm.com*.

The *WSMP* software is packaged into two libraries. The serial and multithreaded single-node routines are a part of the *WSMP* library. This library can be used on single-CPU or SMP workstations. The second library is called *PWSMP* and is meant to be used in the distributed-memory parallel mode.

Note 2.1 *On some platforms, 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 CPU. BLAS calls running in parallel can cause substantial performance degradation.*

Note 2.2 *On many operating 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 and sometimes (for example, on Tru64), changing the limits can be tricky and can require root privileges. You may download the program* memchk.c *from*

http://www.cs.umn.edu/~agupta/wsmp.html *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.*

2.1 AIX on IBM pSeries platform

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

1. XLF Fortran compiler.
2. XLC C language compiler.
3. AIX version 5.1 or higher.

2.1.1 Linking the libraries

The *WSMP* library contains code written in both C and Fortran. The libraries can be linked with either C or Fortran programs using either *xlf_r* or *xlc_r* command.

Compiling your source files in 64-bit mode requires the "-q64" flag and linking to create a 64-bit executable required the "-b64" flag. Also, for 64-bit usage, the environment variable "OBJECT_MODE" must be set to 64.

Here are some examples of linking the user source or object files with the *WSMP* libraries to create an executable file. The user may need to use other flags and options as required by the user's code. Please note that four system libraries, namely, libpthread.a, libhm.a, libhu.a, and libm_r.a are linked via *-lpthread -lhm -lhu -lm_r*. The pthread library provides the *Pthreads* API, *-lhm* and *-lhu* provide fast memory allocation and deallocation functions (which are particularly important for the indefinite, unsymmetric, and iterative solvers), and *-lm_r* provides the reentrant math functions library.

1. 32-bit serial/SMP mode:

   xlc_r <source/object files> -o <executable target> -lpthread -lhm -lhu -lm_r -lxlf90_r -lwsmp -L<path to parent directory of *libwsmp.a*> -bmaxdata:2000000000

2. 64-bit serial/SMP mode:

   xlf_r <source/object files> -o <executable target> -q64 -b64 -lpthread -lhm -lhu -lm_r -lwsmp64 -L<path to parent directory of *libwsmp64.a*>

3. In 64-bit message-passing mode:

   mpxlf_r <source/object files> -o <executable target> -q64 -b64 -lhm -lhu -lm_r -lpthread -lpwsmp -L<path to parent directory of *libpwsmp64.a*>

   mpcc_r <source/object files> -o <executable target> -q64 -b64 -lxlf90_r -lhm -lhu -lm_r -lpthread -lpwsmp -L<path to parent directory of *libpwsmp64.a*>

Note 2.3 *The use of* -lhm -lhu *while linking increases the speed of dynamic memory allocation and usually improves the performance of the solver.*

Note 2.4 *In the 32-bit mode, you will need to use the* -bmaxdata:<*membytes*> *option while linking so that the data segment of the program can use all the available memory. Here* membytes *is the integer number of bytes of memory that you want your program to be able to allocate. If you get an error code which, modulo 1000, is* -102 *(see description of* IPARM(64) *in Section 3.4.9 for more details) when you are not expecting the program to run out of memory, then using this option or increasing* membytes *may solve the problem.*

Note 2.5 *An error code of* -102 *in 64-bit mode can also often be fixed by using an appropriate value of* membytes *in* -bmaxdata:<*membytes*> *while linking.*

Note 2.6 *The reentrant versions of all run-time libraries must be used, even if there is only 1 CPU in your workstation or in each node of your SP. Therefore, only the* xlf_r, xlc_r, mpxlf_r, *or* mpcc_r *commands should be used for linking.*

Note 2.7 *Note that the XLF Fortran library must be linked by using* -lxlf90_r *of* -lxlf90 *if the* xlc_r *command is used for creating the executable. The math library should also be linked in by adding* -lm_r *to the linking command as shown above.*

2.1.2 Setting environment variables and running

In a shared memory parallel environment, the user may want to experiment with setting different values for YIELDLOOPTIME and SPINLOOPTIME to see which values yield the best performance on their system. Generally recommended values are YIELDLOOPTIME = 200 and SPINLOOPTIME = 200, but optimal values may vary depending on the type of machine and the number of CPUs. In most cases, *WSMP* will not yield good performance with the default setting on your system and YIELDLOOPTIME and SPINLOOPTIME must be set to appropriate values. The optimum values of YIELDLOOPTIME and SPINLOOPTIME may be different for the symmetric and unsymmetric direct solvers and may be different for the iterative and direct solvers.

Also in a shared memory parallel environment, the environment variables AIXTHREAD_COND_DEBUG and AIXTHREAD_MUTEX_DEBUG must be set to OFF. Their default value is ON, which may negatively affect the performance of the unsymmetric direct solver.

Finally, you must set the environment variable RT_GRQ to ON to obtain the best possible parallel speedup from multithreading.

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

ksh	**csh** or **tcsh**
export MP_EUILIB=us	setenv MP_EUILIB us
export MP_EUIDEVICE=css0	setenv MP_EUIDEVICE css0
export MP_CSS_INTERRUPT=yes	setenv MP_CSS_INTERRUPT yes
export MP_WAIT_MODE=nopoll	setenv MP_WAIT_MODE nopoll
export MP_PULSE=0	setenv MP_PULSE 0

Note that all *WSMP* and *PWSMP* libraries use Pthreads to exploit possible SMP parallelism on each node. It is usually best to use fewer processes on each node than the number of CPUs per node. If you are using multiple processes per node, please be aware that each process will attempt to use all the CPUs on a node. See Section 4.4 for more details on selecting the number CPU's to be used by each process. Also, while running multiple MPI processes per node, setting the environment variable MP_SHARED_MEMORY to *yes* improves message-passing performance between processes running on the same node. If you are using *PWSSMP* on a shared-memory machine that does not have a high-speed switch installed, then MP_EUILIB and MP_EUIDEVICE can be set to *ip* and *en0*, respectively.

Note that the best values of the MPI related environment variables may be different for different solvers. For example, in addition to the environment variables mentioned in the table above, setting MP_CSS_INTERRUPT to *no* improves the performance of the symmetric solver, and setting it to *yes* improves the performance of the unsymmetric solver.

2.1.3 Linking and execution-time problems

Please refer to Section 2.1.1 for instructions on linking the *WSMP* library. The most likely cause of any problems encountered during linking or loading would be an incompatible version of the operating system or the compiler libraries.

2.2 Linux on IA32 platform

The IA32 Linux version of *WSMP* runs on Intel Pentium 4 and Xeon workstations and clusters. Either Intel's Fortran compiler *ifort* or the GNU C compiler *gcc* can be used to link with the *WSMP* libraries.

2.2.1 Linking the libraries

WSMP for IA32 comes with built-in ATLAS BLAS (*http://math-atlas.sourceforge.net*). The users can link in an optional faster BLAS library such as GOTO (*http://www.cs.utexas.edu/users/flame/goto*) or MKL (*http://www.intel.com/software/products/mkl*). Examples of linking with *WSMP* using the Intel Fortran compiler (use the library in *wsmp-Linux/lib/IA32/Intel* directory) and *gcc* (use the library in *wsmp-Linux/lib/IA32/GNU* directory) are as follows:

ifort -o <executable> <user source or object files> [<optional fast BLAS library>] -lwsmp -L<path of *libwsmp.a*> -lpthread

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

Note that if a fast BLAS library is linked, then the order in which the *WSMP* and the BLAS libraries appear in the linking command is important. If the BLAS library is in the shared-object form (with a .so extension), then it usually must precede -l<*WSMP* library> while linking. If the BLAS library is a static library (with a .a extension), then it usually must appear after -l<*WSMP* library> while linking. Also, please read Note 2.1 carefully.

A single message-passing library *libpwsmp.a* is available for IA32-based Linux clusters. An example of linking on a cluster with IA32 nodes is as follows:

mpif90 -o <executable> <user source or object files> [<optional fast BLAS library>] -lpwsmp -L<path of *libpwsmp.a*> -lpthread

2.2.2 Setting environment variables and running

On all linux platforms, the environment variable MALLOC_TRIM_THRESHOLD_ must be set to -1 and the environment variable MALLOC_MMAP_MAX_ must be set to 0.

The serial and multithreaded version for Linux generally runs well without any special settings. For the message-passing version, please refer to the MPICH documentation available at *http://www-unix.mcs.anl.gov/mpi/mpich/docs.html* and other sources. If you have a cluster with Myrinet or other high-speed network, then you would get much better scalability with a vendor supplied version of MPICH, such as MPICH-GM (*http://www.myri.com/scs/*).

2.2.3 Linking and execution-time problems

Since MPICH is not thread-safe, while running on a cluster or message-passing parallel computer, you must use only one thread per process. You can use the *WSETMAXTHRDS* routine (Section 4.4) to make sure that each process uses only one thread.

Additionally, please refer to Note 2.1.

2.3 Linux on IA64 and x86_64 platforms

Everything from Section 2.2 applies to the 64-bit mode, except that the relevant library names are *libwsmp64.a* and *libpwsmp64.a*. On IA64 and x86_64 (which includes AMD 64-bit processors), these libraries can be linked using Intel's *ifort* compiler, and with Intel's *MKL* library for supplying the BLAS (please see Note 2.1). An example of linking on an IA64 machine is as follows:

ifort -o <executable> <user source or object files> -L<path to Intel MKL> -lmkl_ipf -lguide -L<path of *libwsmp64.a*> -lwsmp64 -lpthread

Both *libwsmp64.a* and *libpwsmp64.a* libraries for linking with *gcc* are alos available in the *wsmp-Linux/lib/X86_64/GNU* directory of the distribution. An example of linking with *gcc* is as follows:

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

Note that on all linux platforms, the environment variable MALLOC_TRIM_THRESHOLD_ must be set to -1 and the environment variable MALLOC_MMAP_MAX_ must be set to 0.

2.4 Linux on AMD processors

The IA32 libraries can be used on 32-bit AMD processors. Separate libraries are provided for 64-bit AMD machines for use with PGI (Portland Group), Intel, and Absoft/GNU compilers. The PGI libraries must be linked using PGI compilers only. The Intel libraries must be linked using the *ifort* compiler. The Absoft/GNU libraries can be linked using either the Absoft compiler or *gcc*. Note that the Absoft/GNU libraries follow the *g77* convention for mangling Fortran subroutine and function names; i.e., a trailing underscore is appended to each name, unless the name contains an underscore anywhere in it, in which case, two trailing underscores are appended (this is true for MPI calls as well). The PGI and Absoft/GNU libraries for AMD processors include ATLAS BLAS; however, the users may be able to improve performance by using the ACML library or GOTO BLAS (please see Note 2.1).

When using the PGI libraries, you must compile your part of the code with the "-tp amd64" option. Examples of linking the serial/multithreaded and the message-passing PGI libraries on 64-bit AMD machines are as follows:

pgf90 -o <executable> <user source or object files> -lwsmp64 -L<path of *libwsmp64.a*> -lpthread

pgf90 -o <executable> <user source or object files> -lpwsmp64 -L<path of *libpwsmp64.a*> -lpthread -lmpich -lfmpich

Note that on all linux platforms, the environment variable MALLOC_TRIM_THRESHOLD_ must be set to -1 and the environment variable MALLOC_MMAP_MAX_ must be set to 0.

2.5 SGI Altix on IA64 processors

The libraries work just the way they do on Linux on IA64 and the Linux license file works on Altix. In the context of *WSMP*, the main difference between Altix and Linux on IA64 is that the MPI libraries are different.

2.6 Other platforms

Currently, the serial/multithreaded libraries are available for the SUN, HP PA-RISC, and Compaq(HP) Alpha machines. In addition, the message-passing library is available for the Compaq(HP) Alpha machines. If you need the *WSMP* or *PWSMP* libraries library for any other platform (with the exception of Microsoft Windows), 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@watson.ibm.com* to discuss this possibility.

2.6.1 SunOS on UltraSPARC or Opteron

Just like on other platforms, *WSMP* software detects the number of CPUs automatically and uses threads to run in parallel on all available CPUs. However, on large SMPs, you may want to limit the number of CPUs that *WSMP* uses because the excessive overhead of using too many threads may actually slow down the application and unnecessarily increase the load on the machine, interfering with other users. Please refer to Section 4.4 for the description of the *WSETMAXTHRDS* function that can be used to control the number of threads.

Currently, only the 64-bit serial/multithreaded library *libwsmp64.a* is available for SunOS machines. The library includes BLAS routines and no external BLAS is necessary. In fact, the use of the *sunperf* library for supplying BLAS is not recommended when using more than one CPU because it does not appear to be thread safe.

The *WSMP* library must be linked with your application as follows:

f90 -o <executable> <user source or object files> -xarch=<architecture> -L<path of *libwsmp64.a*> -lwsmp64 -lpthread -lm -lrt

Please note that the use of *-xarch* option during linking is important. For UltraSPARC processors, *v9* must be used and for AMD processors, *amd64* must be used in the architecture field of this option. In addition, *-xchip=opteron* is also recommended for Opteron processors.

The message-passing library *libpwsmp64.a* is available for SunOS on Opteron CPUs only. It can be linked as follows:

mpif90 -o <executable> <user source or object files> -xarch=amd64 -xchip=opteron -L<path of *libpwsmp64.a*> -lpwsmp64 -lpthread -lm -lrt

2.6.2 Tru64 Unix on Compaq(HP) Alpha processors

When running *WSMP* on Compaq(HP) Alpha processors with Tru64 Unix, care must be taken to avoid underflows, because on this platform, they result in a *floating exception*. Therefore, if the sparse matrix contains entries with very small magnitudes, then these must be explicitly set to zero before passing the matrix to any of the computational routines of *WSMP*. For example, the matrix *invextr1* contains entries with exponents in the range of -300. Unless these entries are set to 0s, a *floating exception* will result. Additionally, for optimum performance on large SMPs, please refer to the description of *WSETMAXTHRDS* in Section 4.4.

The library *libwsmp64.a* needs to be linked as follows:

cc -o <executable> <user source or object files> -L<path of *libwsmp64.a*> -lwsmp64 -pthread -ldxml -lpthread -lm -lfor

It can also be linked using the Fortran compiler as follows:

f90 -o <executable> <user source or object files> -L<path of *libwsmp64.a*> -lwsmp64 -pthread -ldxml -lpthread -lm

Note that you must use *-nofor_main* flag while linking using *f90* when your main program is in C.

A message-passing library *libpwsmp64.a* is available for multiprocessor Tru64 machines for use with Compaq MPI. An example of linking with this library is as follows:

f90 -o <executable> <user source or object files> -L<path of *libpwsmp64.a*> -lpwsmp64 -pthread -ldxml -lpthread -lm -lfmpi -lmpi -lrt

Note 2.8 *The MPI implementation on Tru64 is not thread-safe. Therefore, when using the message-passing library* libpwsmp64.a*, it is mandatory to call* WSETMAXTHRDS *to set the number of threads to 1.*

3 Description of Functionality

WISMP is the primary routine for iterative solution of sparse linear systems and related computations. *WISMP* can work either on a single CPU or on multiple CPUs with a shared address space. This routine can be used for solving sparse linear systems using either direct factorization or preconditioned Conjugate Gradient or GMRES algorithms, for performing sparse matrix-vector and sparse matrix-dense matrix multiplication, and for generating and solving with respect to incomplete factorization based preconditioners. When using *WISMP*, *IPARM(2)* and *IPARM(3)* control the subset of the tasks to be performed (see Section 3.4 for more details).

There six basic tasks that can be performed by calling the *WISMP* routine are summarized in Table 1 and described below:

1. Task 1, Symbolic Analysis: During this step, *WISMP* primarily analyzes the nonzero pattern of the coefficient matrix and performs compression, reordering, and partitioning. While the emphasis is on the structure of the matrix, *WISMP* does look at the current values of the matrix in order to choose appropriate heuristics that are used in this step. This step can usually be followed by multiple steps of solving systems with the same nonzero pattern but different values. Depending on the application, either a single symbolic analysis step may suffice for

Task 1:	Analyze structure and a sample of values of A
Task 2:	Analyze current values of A
Task 3:	Compute Preconditioner M^{-1} such that $M^{-1} \approx A^{-1}$
Task 4:	Iteratively Solve $AX = B$ (approximately compute X = $A^{-1}B$)
Task 5:	Multiply sparse matrix with dense vector/matrix (compute $B = AX$)
Task 6:	Solve w.r.t. preconditioner (compute $X = M^{-1}B$)

Table 1: The tasks that *WISMP* can perform depending on the inputs *IPARM(2)* and *IPARM(3)*.

all matrices with the same structure, or this step may need to be repeated after certain intervals as the values of the matrix become more and more different from the ones considered during this analysis phase. Although, the user has the option of performing symbolic analysis as frequently as desired, *WISMP* usually automatically reanalyzes the matrix when necessary without user intervention.

2. Task 2, Value Analysis: This step actually loads the matrix into internal data structures for subsequent preconditioner generation and (Task 3) and matrix-vector multiplication (either as a part of the iterative solution in Task 4, or as the stand-alone Task 5).

3. Task 3, Preconditioner Generation: This step computes a preconditioner M^{-1} that is close to the inverse of the original coefficient matrix. This preconditioner is subsequently used either as a part of the iterative solution in Task 4, or as the stand-alone Task 6.

   Please refer to Note 3.4 in the description of *IPARM(15)* in Section 3.4.12 for some important information related to this task.

4. Task 4, Solution: This step computes an approximation to X for a linear system $AX = B$, where A is an $n \times n$ nonsingular matrix and X and B are $n \times m$ matrices, $n \geq 1, m \geq 1$. Variations of either GMRES or Conjugate Gradient algorithms are used for obtaining the solution.

5. Task 5, Sparse Matrix-Vector Multiplication: *WISMP* can be used to multiply an sparse matrix with a vector or a with a dense vector or matrix.

6. Task 6, Preconditioning Step: This task can be used to compute computing $X = M^{-1}B$, where M^{-1} is a previously computed preconditioner derived from A (using Task 3).

The same routine, *WISMP*, can perform any of these functions or any valid sequence of these functions depending on the options given by the user via parameter *IPARM* (see Section 3.4). In addition, a call to *WISMP* can be used to get the default values of the options without any of the three basic tasks being performed. See the description of *IPARM(1)*, *IPARM(2)*, and *IPARM(3)* in Section 3.4.12 for more details.

When solving a series of sparse systems with gradually varying coefficient matrices, as is the case in many applications, *WISMP* permits the use of a complete or an incomplete factorization of one matrix to be used as a preconditioner for subsequent systems without computing a new factorization. This feature often results in significant time savings in many applications. By an appropriate choice of *IPARM(15), DPARM(14)*, and *DPARM(15)*, Task 3 can used to generate an exact direct factorization and Task 4 can use used to solve systems with respect to either the factored matrix (when the solution is obtained in a single step), or iteratively with respect to a previously factored nearby matrix. Please refer to the description of *IPARM(2), IPARM(15), DPARM(14), DPARM(15)* for more details.

Note that the data-structures generated as a result of the analysis of A during Tasks 1 and 2 and the preconditioner M generated during Task 3 are stored internally and are not directly accessible to the user.

The *WISMP* routine performs 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 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.

3.1 Types of matrices accepted and their input format

WISMP accepts the sparse coefficient matrices in the compressed sparse row (CSR) or compressed sparse column (CSC) formats. For symmetric matrices, the user has the option of either proving the full matrix as input, or just a triangular portion. Figure 1 illustrates both the full and the triangular storage formats.

WISMP 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 3.4.12 for more details.

3.2 Krylov subspace solvers supported

WISMP currently includes implementations of preconditioned conjugate gradient method and GMRES. We expect to include more solvers in the future. *WISMP*'s current focus is on robust high-performance preconditioners. As described in the beginning of Section 3, users can call *WISMP*'s preconditioner generation (Step 3) and application (Step 6) steps, as well as sparse matrix-vector multiplication (Step 5) in their own implementations of a solver (they must be preceded by Step 1 for each new structure and Step 2 for each set of new values of the sparse matrix).

The type of solver can either be chosen by the user (see *IPARM(13)*), or can be picked automatically by *WISMP* depending on the type of input matrix and the preconditioner.

3.3 Preconditioners and their parameters

WISMP currently supports Jacobi (diagonal), Gauss-Siedel (SSOR with relaxation $\omega = 1$), Incomplete Cholski, and Schur-complement based multilevel preconditioners for symmetric positive definite and mildly indefinite (with very few negative eigenvalues) matrices. It supports Jacobi, Gauss-Siedel, and incomplete LU factorization based preconditioners for general matrices.

For preconditioners based on incomplete factorization, the choice of two thresholds, τ (drop tolerance) and γ (fill factor), is critical to the performance and convergence of the solver. *WISMP* gives users the option of either letting it determine and tune these thresholds automatically, or using fixed user-determined values. Please refer to the descriptions of *IPARM(15), IPARM(34), DPARM(14),* and *DPARM(15)* for more details. In an application that involves repeated preconditioner generation and solution w.r.t. gradually varying coefficient matrices, *WISMP*, if given the option, *WISMP* can automatically tune these parameters to suitable values and even dynamically adjust them to fit shifting spectral properties of the coefficient matrices. When solving only one system at a time, *WISMP* outputs suggested fixed values of the thresholds that the user can use in subsequent runs to improve performance.

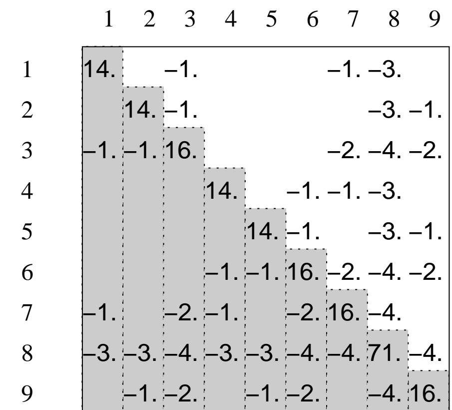
3.4 Calling sequence of the WISMP subroutine

There are five types of arguments, namely input (type **I**), output (type **O**), modifiable (type **M**), temporary (type **T**), and reserved (type **R**). The input arguments are read by *WISMP* 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 *WISMP* and modified to return some information, the temporary arguments are not read but their contents are overwritten by unpredictable values during execution, and the reserve arguments are just like temporary arguments which may change to one of the other types of arguments in the future serial and parallel releases of this software.

In the remainder of this document, the "system" refers to the sparse linear system of N equations of the form $AX = B$, where A is a sparse symmetric coefficient matrix of dimension N, B is the right-hand-side vector/matrix and X is the solution vector/matrix, whose approximation $\overline{X}$ is computed by *WISMP*.

©IBM Corporation 1997, 2007. All Rights Reserved.

K	CSR-UT Format			CSR-full Format		
	IA(K)	JA(K)	AVALS(K)	IA(K)	JA(K)	AVALS(K)
1	1	1	14.0	1	1	14.0
2	5	3	-1.0	5	3	-1.0
3	9	7	-1.0	9	7	-1.0
4	13	8	-3.0	15	8	-3.0
5	17	2	14.0	19	2	14.0
6	21	3	-1.0	23	3	-1.0
7	25	8	-3.0	29	8	-3.0
8	27	9	-1.0	35	9	-1.0
9	29	3	16.0	44	1	-1.0
10	30	7	-2.0	50	2	-1.0
11		8	-4.0		3	16.0
12		9	-2.0		7	-2.0
13		4	14.0		8	-4.0
14		6	-1.0		9	-2.0
15		7	-1.0		4	14.0
16		8	-3.0		6	-1.0
17		5	14.0		7	-1.0
18		6	-1.0		8	-3.0
19		8	-3.0		5	14.0
20		9	-1.0		6	-1.0
21		6	16.0		8	-3.0
22		7	-2.0		9	-1.0
23		8	-4.0		4	-1.0
24		9	-2.0		5	-1.0
25		7	16.0		6	16.0
26		8	-4.0		7	-2.0
27		8	71.0		8	-4.0
28		9	-4.0		9	-2.0
29		9	16.0		1	-1.0
30					3	-2.0
31					4	-1.0
32					6	-2.0
33					7	16.0
34					8	-4.0
35					1	-3.0
36					2	-3.0
37					3	-4.0
38					4	-3.0
39					5	-3.0
40					6	-4.0
41					7	-4.0
42					8	71.0
43					9	-4.0
44					2	-1.0
45					3	-2.0
46					5	-1.0
47					6	-2.0
48					8	-4.0
49					3	16.0



A 9 X 9 symmetric sparse matrix.

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

Figure 1: Illustration of the upper triangular CSR and full CSR formats for a symmetric matrix for the serial/multithreaded *WISMP* routine.

Note 3.1 *Recall that* WISMP *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* WISMP.

Note 3.2 *The original code for* WISMP *is in Fortran and expects the parameters to be passed by reference. Therefore, when calling* WISMP *from a C program, the addresses of the parameters described in Section 3.4 must be passed.*

The calling sequence and description of the parameters of *WISMP* is as follows. Note that all arguments are not accessed in all phases of the solution process. The descriptions that follow indicate when a particular argument is not accessed. When an argument is not accessed, a NULL pointer or any scalar can be passed as a place holder for that argument. The example program *wismp_ex1.f* at the *WSMP* home page illustrates the use of the *WISMP* subroutine for the matrix shown in Figure 1.

WISMP (N, IA, JA, AVALS, B, LDB, X, LDX, NRHS, RMISC, CVGH, IPARM, DPARM)

3.4.1 N (type I): matrix dimension

```
INTEGER N
```

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

3.4.2 IA (type I): row/column pointers

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

IA is an integer array of size one greater than N. *IA(I)* points to the first row/column index of column/row I in the array *JA*. Refer to Figure 1 and description of *IPARM(4)* in Section 3.4.12 for more details.

3.4.3 JA (type I or M): column/row indices

```
INTEGER JA ( * )
```

The integer array *JA* contains the column (row) indices of the upper (lower) triangular part of the symmetric sparse matrix A. The indices of a column (row) are stored in consecutive locations. In addition, these consecutive column (row) indices of a row (column) *must* be sorted in increasing order upon input. *WSMP* provides two utility routines to sort the indices (see Section 4 for details). The size of array *JA* is the total number of nonzeros in the matrix A or one of its triangular portions (including the diagonal) if A is symmetric and *IPARM(4)* is 2 or 3.

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

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

The array *AVALS* contains the actual double precision values corresponding to the indices in *JA*. The size of *AVALS* is the same as that of *JA*. See Figure 1 for more details. Please note that, in many cases, *AVALS* is accessed during Task 1 (see description of *IPARM(2..3)* in Section 3.4.12), whose primary aim is analyze the structure of the matrix. The reason for accessing *AVALS* during this phase is to ensure that *WISMP* does not perform any structural rearrangement of the matrix that may turn out to be numerically unstable in subsequent phases. Therefore, *AVALS* must be available to *WISMP* even in the structural analysis phase.

Note 3.3 *By default,* IPARM(14) *is 1 (see description of* IPARM(14)*), which means that* AVALS *is overwritten during preconditioner generation and must be passed unaltered to the solution phase when preconditioner type (*IPARM(15)*) is greater than or equal to 3. This can be switched off and* AVALS *can be made read-only by setting* IPARM(14) *to 0.*

3.4.5 B (type I or O): right-hand side vector/matrix

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

The $N\times$ *NRHS* dense matrix B (stored in an *LDB* $\times$ *NRHS* array) contains the right-hand side of the system of equations $AX = B$ to be solved. If the number of right-hand side vectors, *NRHS*, is one, then B can simply be a vector of length N. The input B is accessed only in the solution phase (Task 4).

When calling *WISMP* for multiplying a sparse matrix with a dense vector or matrix (task 5), then B is the output product vector or matrix.

3.4.6 LDB (type I): leading dimension of B

```
INTEGER LDB
```

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

3.4.7 X (type O or I): solution vector/matrix

```
DOUBLE PRECISION X ( LDX, NRHS )
```

The $N\times$ *NRHS* dense matrix X (stored in an *LDX* $\times$ *NRHS* array) contains the computed solution the system of equations $AX = B$ if task 3 is performed by *WISMP*. If the number of right-hand side vectors, *NRHS*, is one, then X can simply be a vector of length N.

When *WISMP* is called for multiplying a sparse matrix with a dense vector or matrix (task 5), then X is the input vector or matrix.

3.4.8 LDX (type I): leading dimension of X

```
INTEGER LDX
```

LDX is the leading dimension of the solution matrix if *NRHS* > 1. When used, *LDX* must be greater than or equal to N. Even if *NRHS* $= 1$, *LDX* must be greater than 0.

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

```
INTEGER NRHS
```

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

3.4.10 RMISC (type I, O, M): double precision output info

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

If *IPARM(25)* is 0, then *RMISC* is not accessed. If *IPARM(25)* is 1, then on return from the solution phase, *RMISC(I,J)* is set to to the I-th component of the residual while solving for the J-th RHS. If *IPARM(25)* is 2 on input, then the contents of *RMISC* are used as the starting approximation to the solution of the system. By default, the starting solution contains all zeros. However, if the user has access to a good approximation, then using that may require fewer iterations for convergence. If *IPARM(25)* is 3, then the contents of *RMISC* are used as the starting solution on input and are overwritten by the residual on output.

Note that the user needs to provide a valid double precision array of size $N \times NRHS$ only if *IPARM(25)* is set to a nonzero value; otherwise, *RMISC* can just be a NULL pointer.

3.4.11 CVGH (type O): double precision convergence history output

```
DOUBLE PRECISION CVGH ( 0 : IPARM(6) )
```

Please refer to the description of *IPARM(27)*. The user needs to provide a valid double precision array of size *IPARM(6)* + 1 only if *IPARM(27)* is set to a nonzero value; otherwise, *CVGH* can just be a NULL pointer.

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

```
INTEGER IPARM ( 64 )
```

IPARM is an integer array of size 64 that is used to pass various optional parameters to *WISMP* and to return some useful information about the execution of a call to *WISMP*. If *IPARM(1)* is 0, then *WISMP* 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 2. *IPARM(1)* through *IPARM(3)* are mandatory inputs, which must always be supplied by the user. If *IPARM(1)* is 1, then *WISMP* 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 *WISMP* 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(37:63)* must be filled with 0's.

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

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

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

  As described in Section 3, the *WISMP* routine can be used to perform a subset of several tasks listed in in Table 1. This subset of tasks performed is controlled by *IPARM(2)* and *IPARM(3)*.

  On input, *IPARM(2)* must contain the number of the starting task and *IPARM(3)* must contain the number of the last task. All tasks from *IPARM(2)* to *IPARM(3)* are performed, provided that it is a valid task sequence. Table 3 shows all sets of valid groups of tasks that can be performed by setting *IPARM(2)* and *IPARM(3)* appropriately.

  If *IPARM(2)* $\leq$ 0 or *IPARM(2)* $>$ 6 or *IPARM(2)* $>$, then no tasks are performed; however, if *IPARM(1)* is 0, then *IPARM(4)* to *IPARM(64)* and *DPARM(4)* to *DPARM(64)* are filled with default values.

  On output, *IPARM(2)* contains 1 + number of the last task performed by *WISMP*, if any. This is to facilitate users to restart processing on a problem from where the last call to *WISMP* left it. Also, if *WISMP* is called to perform multiple tasks in the same call and it returns with an error code in *IPARM(64)*, then the output in *IPARM(2)* indicates the task that failed. If *WISMP* performs no task, then, on output, *IPARM(2)* is set to *max(IPARM(2),IPARM(3)+* 1*)*.

  When using *WISMP* to solve a single system of equations, you would perform Tasks 1–4 in sequence. When solving multiple systems with the same coefficient matrix A, but different RHS vectors/matrices B, you would perform Tasks 1–3, followed by multiple calls to *WISMP* to perform Task 4. Note that if all the RHS vectors are available at the same time, then it is much more efficient to perform a single instance of Task 4 with all of them bundled in the matrix B and setting *NRHS* to the number of RHS vectors (which is the number of columns in B). When solving multiple systems with different coefficient matrices, you would perform Tasks 1–4 for the first system, followed by Tasks 2–4 or just Tasks 2 and 4 for the subsequent systems. If the coefficient matrices of the consecutive systems to be solved change gradually, then Task 3 may not be needed for each system. This may gradually increase the number of iterations required by Task 4 to solve the system because the stale preconditioner

	IPARM			DPARM		
Index	Default	Description	Type	Default	Description	Type
1	mandatory I/P	default/user defined	M	-	unused	-
2	mandatory I/P	starting task	M	-	unused	-
3	mandatory I/P	last task	I	-	unused	-
4	0	I/P format	I	-	max. fact. diag.	O
5	1	numbering style	I	-	min. fact. diag.	O
6	1000	max iterations	I	1×10^{-8}	target rel. resid. norm	I
7	0	matrix type	I	1×10^{-5}	target rel. error norm	I
8	0	max. matching use	I	0.0	termination criterion	I
9	10	max. matching frequency	I	-	unused	-
10	1	scaling option	I	-	unused	-
11	2	thresh. pivoting opt.	I	1×10^{-3}	pivot thresh.	I
12	1	pivot perturb. opt.	I	2×10^{-8}	small piv. thresh.	I
13	0	solver choice	I	-	no. of supernodes	O
14	1	AVALS/JA reuse opt.	I	0.0	threshold τ	M
15	3	preconditioner choice	I	0.0	threshold γ	M
16	1	ordering option 1	I	-	unused	-
17	0	ordering option 2	I	-	unused	-
18	3	max imbalance	I	-	unused	-
19	0	refinement strategy	I	-	unused	-
20	0	matrix characteristics	I	-	unused	-
21	0	GMRES restart	I	-	structural symmetry	O
22	0	# approx. eigenvecs	I	2×10^{-8}	small piv. repl.	I
23	-	incomplete fact. size	O	-	incomplete fact. ops.	O
24	40	max. clique size	I	0.8	min. compression efficiency	I
25	0	RMISC use	I	-	unused	-
26	-	number of iterations	O	-	relative residual norm	O
27	0	CVGH use	I	-	relative error norm	O
28	0	max inner iterations	I	0.0	inner residual	I
29	3	inner preconditioner	I	-	unused	-
30	-	# diags replaced/interchanges	O	-	unused	-
31	-	# entries dropped	O	-	unused	-
32	1	precond. repeat use	I	-	unused	-
33	-	no. of CPU's used	O	-	unused	-
34	2	τ, γ tuning	I	-	unused	-
35	1	mat-vec tuning	I	-	unused	-
36	1	automatic reattempt	I	-	unused	-
37-63	0	reserved	R	0.0	reserved	R
64	-	return error code	O	-	unused	-

Table 2: The default initial values of the various entries in *IPARM* and *DPARM* arrays. A '-' indicates that the value is not read by *WISMP*. Please refer to the text for details on ordering options *IPARM(16:20)*.

IPARM(2)	*IPARM(3)*
0	0, 1, 2, 3, 4
1	1, 2, 3, 4
2	2, 3, 4
3	3, 4
4	4
5	5
6	6

Table 3: Valid values of *IPARM(2)* (first column) and the corresponding valid values of *IPARM(3)*.

Input matrix format	*IPARM(4)*
Compressed Sparse Rows (CSR); full matrix	1
Compressed Sparse Columns (CSC); full matrix	2
Upper triangular portion of symmetric matrix in CSR format	3
Lower triangular portion of symmetric matrix in CSR format	4

Table 4: Input formats accepted for sparse coefficient matrices and the corresponding values of *IPARM(4)*.

will be used, but may still results in an overall faster solution because the stale preconditioner may be only slightly worse than the preconditioner computed from the current matrix. The user may perform Task 3 after any number of steps. *WISMP* also keeps track of the relative amounts of computation in Tasks 3 and 4, and automatically recomputes the preconditioner at optimum (possibly varying from one matrix to another) intervals.

The ability to use a preconditioner generated from an earlier coefficient matrix can also be used in conjunction with *WSMP*'s direct solvers to obtain fast solutions to a sequence of linear systems with gradually varying coefficient matrices. As described in Section 3, by choosing appropriate values of *DPARM(14–15)*, Task 3 can be used to perform a direct factorization. In this case, Task 4 obtains the solution in a single step corresponding to the matrix that is factored. However, this factorization can serve as a preconditioner for subsequent systems, which (hopefully) can be solved in a small number of iterations. After a point, the number of iterations in Task 4 may increase to a level that it may worthwhile recomputing the factors using Task 3. Either the user can explicitly perform Task 3; however, if the user does not, then *WISMP* can automatically detect where, if at all, it needs to recompute the preconditioner within a repeated sequence of Tasks 2 and 4.

WISMP also provides users stand-alone access to fast parallel sparse matrix-vector multiplication (or multiplication of a sparse matrix with a dense matrix) and preconditioning. The users can call *WISMP* to provide these functions for their own iterative algorithms other than CG and GMRES currently provided by *WISMP*. In this case, the user would first perform Tasks 1–2 if preconditioning is not used, or Tasks 1–3 if preconditioning is used. These steps can then be followed by multiple calls to *WISMP* to perform Tasks 5 and 6 if the user chooser to implement their own iterative method and use *WISMP*'s fast parallel sparse matrix-vector multiplication and preconditioning algorithms.

Please refer to Note 3.4 in the description of *IPARM(15)* in Section 3.4.12 for some important information related to Task 3.

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

  IPARM(3) must contain the number of the last task to be performed by *WISMP*. In a call to *WISMP*, 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 1–6, then no task is performed. This can be used to fill *IPARM* and *DPARM* with default values; e.g., by calling *WISMP* with *IPARM(1)* = 0, *IPARM(2)* = 0, and *IPARM(3)* = 0.

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

Coefficient matrix type	*IPARM(7)*
Real or complex general unsymmetric	0
Real or complex unsymmetric diagonally dominant	1
Real or complex symmetric indefinite	2
Real or Hermitian symmetric positive-definite	3
Real symmetric M-matrix	4

Table 5: Types of sparse coefficient matrices and the corresponding values of *IPARM(7)*.

IPARM(4) denotes the format in which the coefficient matrix A is stored. Table 4 lists the supported input matrix formats and the corresponding input values for *IPARM(4)*.

The default value of *IPARM(4)* is 1.

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

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

  The default value of *IPARM(5)* is 1.

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

  On input, *IPARM(6)* should be set to the maximum number of iterations of the Krylov subspace method that the user wishes to perform before terminating if the relative residual has not converged to the desired limit (specified in *DPARM(6)*). The actual number of iterations required to reach the desired convergence is returned in *IPARM(26)* after the solve phase of *WISMP*. Please refer to the description of *DPARM(6)* for more details.

  The default input value of *IPARM(6)* is 1000.

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

  IPARM(7) denotes the the type of coefficient matrix. Table 5 lists the types of sparse systems and the corresponding input values for *IPARM(7)*. The default value of *IPARM(7)* is 0. It is extremely important that the matrix type is indicated as accurately as possible in *IPARM(7)*.

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

  IPARM(8) is ignored for matrix types 1, 3, and 4.

  For matrix types 0 and 2, *WISMP* 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 [10, 2, 8, 7]. 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 for non positive-definite matrices, and if *IPARM(8)* is 2, then this permutation is not performed. It is recommended that maximum weight matching be turned off by setting *IPARM(8)* to 2 for diagonally dominant symmetric indefinite matrices.

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

  Depending on the input in *IPARM(8)*, *WISMP* 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. When multiple systems with coefficient matrices of the same structure but gradually varying values are solved, then it may be more economical to not perform the maximum matching each time. The input *IPARM(9)* can be used to control the frequency at which such matching is performed. The default value of *IPARM(9)* is 10; i.e., the maximum bipartite matching algorithm is used for every 10th matrix and for the others, the last computed row permutation and scaling is applied.

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

  An input of *IPARM(10)* = 0 implies that *WISMP* will not scale the input matrix. *IPARM(10)* = 1, which is the default, implies that scaling is performed in an attempt to improve convergence and the numerical properties of incomplete factorization.

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

  IPARM(11) is ignored for matrix types 1, 3, and 4.

  For matrix types 0 and 2, if a preconditioner based on incomplete factorization is chosen, then *IPARM(11)* and *IPARM(12)* instruct *WISMP* 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 *WISMP* 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 *WISMP* 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.

  If *IPARM(11)* = 2 upon input, which is the default, then *WISMP* decides whether or not to perform pivoting, based on the properties of the matrix. It is recommended that pivoting be turned off by setting *IPARM(11)* to 0 for diagonally dominant symmetric indefinite matrices.

  The total number of row or column interchanges performed are reported in *IPARM(30)*.

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

  IPARM(12) is ignored for matrix types 1, 3, and 4. *IPARM(12)* is also ignored if pivoting is performed during incomplete factorization; i.e., either *IPARM(11)* is 1 or *IPARM(11)* is 2 and *WISMP* elects to perform pivoting. *IPARM(12)* = 0 has no effect.

  If *IPARM(12)* is 1, which is the default, and if pivoting is not performed during incomplete factorization, then the following action is taken. Let $\alpha = A(i,i)$ just before the i-th step of incomplete factorization and let $A(l,i)$, $l > i$, be such that $|A(l,i)| > |A(j,i)|$ for all $j > i$. If $|alpha| < |A(l,i)| \times$ *DPARM(12)*, then $A(i,i)$ is replaced by an entry whose sign is the same as α and whose magnitude is $|A(l,i)| \times$ *DPARM(22)*.

  The total number of diagonals perturbed are reported in *IPARM(30)*.

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

  If *IPARM(13)* is 0 (default), the *WISMP* makes the choice of the solver. If *IPARM(13)* is 1, then the conjugate gradient method is used. If *IPARM(13)* is 2, then a GMRES solver used.

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

  IPARM(14) can be used to reuse the space in the arrays *AVALS* and *JA* during the preconditioner generation for preconditioner types 3 and 4 (see description of *IPARM(15)* for details on preconditioner types). Thus, this option can be used to save memory if user does not need access to the values, or indices, or both values and indices of the coefficient matrix after solving the system.

Preconditioner type	*IPARM(15)*
No preconditioning	0
Diagonal preconditioner	1
SSOR preconditioner	2
Incomplete factorization	3
Multilevel preconditioner	4

Table 6: Types of preconditioner supported in *WISMP* and the corresponding iput values in *IPARM(15)*.

If *IPARM(14)* is 0, then *AVALS* and *JA* are not altered by *WISMP*. If *IPARM(14)* is 1, which is the default, then *AVALS* is overwritten during preconditioner generation (Task 3). The *AVALS* after preconditioner generation must be passed unaltered to the solve phase (Task 4). If *IPARM(14)* is 2, then *JA* is overwritten during preconditioner generation (Task 3). If *IPARM(14)* is 3, then both *AVALS* and *JA* are overwritten during Task 3 and must be passed unaltered to Task 4.

IPARM(14) is ignored if *IPARM(15)* is 0, 1, or 2 and *AVALS* is never overwritten for these preconditioner types.

Note that *IPARM(14)* must be set to 0 if a preconditioner generated from one matrix is used to solve a system with a different coefficient matrix while solving a sequence of linear systems because filling *AVALS* and *JA* with the values and indices of the new matrix will corrupt the preconditioner stored in these arrays.

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

  IPARM(15) indicates the type of preconditioning to be performed.

  Table 6 lists the types of preconditioning available in *WISMP* and the corresponding input values for *IPARM(15)*.

  If *IPARM(15)* is 0, then no preconditioning is used. If *IPARM(15)* is 1, then Jacobi preconditioning (diagonal scaling) is used. If *IPARM(15)* is 2, then SSOR preconditioning is performed (with a value of 1.0 for ω). The default value for *IPARM(15)* is 3, which results in an incomplete factorization. *IPARM(15)* = 4 results in a Schur complement based algebraic multilevel preconditioning.

  An incomplete factorization based on two thresholds, τ and γ is used for preconditioners of type 3 and 4. During incomplete factorization, all entries in locations (i, j) and (j, i) whose magnitudes are smaller than or equal to τ times the diagonal entry (i, i) are dropped, where $i > j$. Furthermore, the incomplete factors would contain at most γ times the original number of nonzeros in each row and column, and additional fill may be dropped to satisfy this constraint. The thresholds τ and γ are initially picked based on the user specified inputs in *DPARM(14)* and *DPARM(15)*, respectively. However, during the incomplete factorization, τ may be reduced and γ increased if the factorization algorithm detects that that is necessary to maintain the stability and accuracy of the incomplete factorization.

  Please refer to the description of *DPARM(14)*, *DPARM(15)*, and *IPARM(16)* for more details. Note that *DPARM(14)* and *DPARM(15)* are ignored if *IPARM(15)* is 0, 1, or 2.

  When solving several systems with different coefficient matrices, the user can apply different types of preconditioners to different systems. However, whenever the preconditioner is changed, the user must restart from Task 1 (i.e., perform the symbolic analysis again) because Task 1 uses preconditioner dependent heuristics.

  Note 3.4 *Even if* IPARM(15) *is 0; i.e., no preconditioning is used, Task 3 must be performed, at least once, because some internal data structured required in subsequent tasks are allocated in this step. In other words, even if preconditioning is not used, Task 3 cannot be skipped and must be performed with* IPARM(15) = *0, at least for the first coefficient matrix.*

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

WISMP uses a preordering of the matrix rows and columns to minimize fill during incomplete factorization for preconditioners of type 3 and 4. This reordering is performed as a part of Task 2.

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-KcKee ordering [3] is performed. If *IPARM(16)* is 1, 2, or 3, then a graph-partitioning based ordering [4] is performed. In addition, when *IPARM(16)* is positive, the ordering speed and quality is determined by its integer value. *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 seeking only one or a few solutions per structural analysis step, it is advisable to change *IPARM(16)* to 3 or 2.

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

  If a multilevel or incomplete factorization based preconditioner is used, then *IPARM(17)* can be used to choose partitioning and ordering heuristics based on the numerical values in the coefficient matrix. *WISMP* uses a combination of a few such heuristics. If *IPARM(17)* is 0, which is also the default, then *WISMP* chooses the combination of heuristics automatically, based on the characteristics of the matrix.

  Three other combinations are available and can be selected by setting *IPARM(17)* to 1, 2, or 3, respectively. In addition to the default, users are encouraged to experiment with the other three as well to determine which one works best for their problems.

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

  WISMP uses multilevel graph partitioning algorithms [5] to distribute data and computation among multiple CPUs. The inputs in *IPARM(18:19)* control the partitioning process. *IPARM(18)* specifies the maximum percentage of tolerable load imbalance; default is 3% (i.e., *IPARM(18)* = 3 by default). Note that only whole number percentages for tolerable imbalance can be specified because *IPARM* is an integer array. Imbalance is computed as the ratio of the weight of the heaviest part to average part weight.

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

  IPARM(19) specifies the type of refinement to be used during multilevel graph partitioning. If *IPARM(19)* is 0, then *WISMP* chooses the refinement strategy based on the number of CPU's and the size of the matrix. A value of 11 results in greedy refinement (GR) and a value of 12 results in Kernighan-Lin (KL) refinement. KL is slower than GR, especially for moderate to large number of CPUs. Fortunately, the benefit of using KL over GR is the maximum for small number of partitions, where the run-time penalty is not excessive. For large number of parts, GR does as well as KL and is much faster. For small number of parts relative to the size of the graph, KL is recommended. Please refer to [5] for details on refinement strategies.

  The default value of *IPARM(19)* is 0.

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

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

  Certain sparse matrices have a very irregular structure and have a few rows/columns that are much denser than most of the rows/columns. Many sparse matrices arising from linear programming problems fall in this category. For such matrices, the quality and the speed of partitioning can usually be improved by setting *IPARM(20)* to 1. This instructs the partitioning and ordering routines to split the graph based on the high degree nodes before proceeding.

  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 *WISMP* to construct a compressed graph before proceeding with the partitioning or ordering, which then run much faster as they work on the smaller compressed graph rather than the original larger graph.

The symbolic factorization phase before incomplete factorization (relevant only for preconditioners of type 3 or 4) may fail for some matrices with very irregular structure, unless *IPARM(20)* is set to 1.

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

  IPARM(21) is ignored for the Conjugate Gradient solver. For the GMRES solver, the input in *IPARM(21)* is not 0, then it is used as the restart parameter; i.e., GMRES is restarted after every *IPARM(21)* iterations. If *IPARM(21)* is 0, then *WSMP* chooses the restart parameter based on certain properties of the matrix. The default value of *IPARM(21)* is 0. A small value of the restart parameter may slow down the convergence rate but will reduce memory use and average time per iteration. A higher value may increase the convergence rate at the cost of additional memory and average CPU time per iteration.

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

  IPARM(22) is ignored for the Conjugate Gradient solver. The GMRES method implemented in the *WSMP* library usually adds approximate eigenvectors corresponding to a few smallest eigenvalues of the matrix to the subspace in order to mitigate the impact of restarting on convergence [9]. If the input in *IPARM(22)* is not 0, *IPARM(22)* approximate eigenvectors are used. If *IPARM(22)* is 0, then *WSMP* attempts to choose an appropriate number of eigenvectors. If *IPARM(22)* is -1, then approximate eigenvectors are not used. The default value of *IPARM(22)* is 0.

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

  If an incomplete factorization or multilevel preconditioner is used, then after the preconditioner generation phase, *IPARM(23)* contains the total number of double words required to store the incomplete factors.

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

  In order to improve the efficiency of sparse matrix-vector multiplication and incomplete factorization preconditioning (if opted for), *WISMP* attempts to find groups of rows and columns that have identical nonzero patterns. Each such group is referred to as a *supernode*. It is often beneficial to group even those rows (and columns) into supernode that do not have an identical structure, but whose structures are nearly identical. This is accomplished by introducing artificial entries with a numerical value of zero into these rows and columns to make their structures identical. *IPARM(24)* and *DPARM(24)* are user inputs that can be used to tell *WISP* how aggressively it should introduce these extra entries in order to maximize the size the supernodes.

  IPARM(24) indicates the maximum size of any supernode that would be used by *WISMP*. The naturally occurring supernodes will be restricted in size to a maximum of *IPARM(24)* and when the size of a naturally occurring supernode is less than *IPARM(24)*, then no artificial entries will be used to increase its size beyond *IPARM(24)*.

  DPARM(24), whose default value is 0.8, indicates the minimum fraction of overlap in the nonzero pattern of two row-column pairs needed for them to be a part of the same supernode. of any row and column that must comprise of the original nonzero entries. In other words, the number of extra entries added to a row or column will not exceed (1.0 - *DPARM(24)*) times the original number of entries in that row or column.

  Increasing *IPARM(24)* and decreasing *DPARM(24)* (whose valid range is from 0.0 to 1.0) increases the size and reduces the number of supernodes, while increasing the total number of artificial entries added to the matrix.

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

  Please refer to the description of *RMISC* in Section 3.4.10.

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

  Upon return from the solution phase, *IPARM(26)* contains the total number of iterations performed. It is always less than or equal to the input in *IPARM(6)*.

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

  IPARM(27) = 0, which is the default, has no effect. If *IPARM(27)* = 1 during the solution phase, then the convergence history is returned in *CVGH*; i.e., it contains *IPARM(26)* + 1 double precision values that correspond to the relative residual after each iteration. *CVGH(0)* contains initial relative norm of residual before starting the iterations, and is equal to 1.0 if *IPARM(25)* < 2. If *IPARM(27)* = 1, then *CVGH* must point to a valid user-supplied double precision array of size *IPARM(6)* + 1.

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

  If the preconditioner type in *IPARM(15)* is 4 (multilevel preconditioner), then the input in *IPARM(28)* indicates the maximum number of inner iterations to be performed at any level. The default value of *IPARM(28)* is 0. The inner iterations are terminated before *IPARM(28)* iterations are performed if the norm of the residual at that level drops below a certain threshold. This threshold is chosen based on the input value in *DPARM(28)*. If *DPARM(28)* is 0.0 on input (default) *WISMP* selects an appropriate value for this threshold. If *DPARM(28)* contains a nonzero value, then this value is used as the threshold to terminate the inner iterations. If *IPARM(28)* is 0 (default) then no inner iterations are performed and *DPARM(28)* is ignored.

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

  IPARM(29) specifies the type of preconditioning to used in the inner iterations of a multilevel preconditioner if the main preconditioner (*IPARM(15)*) is of type 4. All preconditioners listed in Table 6 are valid choices for an inner preconditioner. Note that if a multilevel preconditioner (type 4) is chosen for the inner iterations as well and the number if inner iterations specified in *IPARM(28)* is greater than 0, then preconditioned iterations are performed recursively at each level, which may consume an impractically high CPU time.

- **IPARM(30), type O:**

  If replacement of very small diagonal entries is selected by using *IPARM(12)* = 1 for incomplete factorization or multilevel preconditioners, then at the end of the preconditioner generation step, *IPARM(30)* contains the number of such replacements performed. Alternatively, if pivoting is selected by using *IPARM(11)* = 1 during incomplete LU factorization, then at the end of the preconditioner generation step, *IPARM(30)* contains the number of row and column interchanges.

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

  At the end of the preconditioner generation step, *IPARM(31)* contains the total number of entries dropped during incomplete factorization or multilevel preconditioner computation.

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

  The input in *IPARM(32)* specifies the number of times the preconditioner is expected to be reused. *WISMP* uses this information to spend the appropriate amount of effort in generating the preconditioner. Note that *IPARM(32)* must refer to the expected number of times that *WISMP* would be called for an iterative solution after generating a preconditioner, and must be independent of *NRHS* in these calls.

  The default value of *IPARM(32)* is 1.

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

  On output, *IPARM(33)* is set to the number of CPU's that were used on the node/workstation in SMP mode. This is the number of CPU's physically present on the node/workstation, unless it is overridden by a call to the *WSETMAXTHRDS* routine (Section 4.4).

- **IPARM(34), type I:**

  If *both* inputs *DPARM(14)* and *DPARM(15)* are 0.0, and the preconditioner type (*IPARM(15)*) is 3 or 4, then the solver enters what we will refer to as the *automatic threshold tuning mode* for solving multiple systems of equations involving repeated preconditioner generation and iterative solution cycles. This scenario is frequently

encountered in many applications, including those that involve solving a non-linear system. By default, this mode is turned on because the default values of *DPARM(14)* and *DPARM(15)* are 0.0. In this mode, *WISMP* chooses appropriate values for τ and γ (see description of *IPARM(15)* for more details) and refines them from iteration to iteration in order to balance the cost of preconditioner generation and iterative solution so that the overall solution time is optimized. *WISMP* uses both timing and numerical criteria to adjust the thresholds between iterations.

While this technique is quite effective in optimizing the solution time, a somewhat undesirable side-effect of using it is that the results may not be replicatable from one run to another for the same input data due to variation in the timings of various steps in different runs. Therefore, *WISMP* provides users the option to use automatic threshold tuning with varying degrees of aggressiveness. Setting *IPARM(34)* to 0 forces *WISMP* to completely disregard timing information and use only numerical criteria for the selection and modification of the incomplete factorization thresholds. Thus, setting *IPARM(34)* to 0 guarantees the same solution for the same problem each time. Other valid values of *IPARM(34)* are 1, 2, and 3. A higher value of *IPARM(34)* results in a higher degree of variation in results between different runs, but usually also an overall faster solution.

The default value of *IPARM(34)* is 2.

- **IPARM(35), type I:**

  WISMP may chose an appropriate blocking factor for sparse matrix-vector multiplication by attempting different block sizes and monitoring the computation times during the first few iterations. While this improves the efficiency of the subsequent iterations, sometimes, this may make it hard to replicate the exact results with the same data from one run to another. The reason is that the variations in timing between different runs may result in the selection of different block sizes.

  The default value of *IPARM(35)* is 1, which enables this optimization. It can be switched off by setting *IPARM(35)* to 0, in which case, a fixed block size will be used.

- **IPARM(36), type I:**

  IPARM(36) = 0 has no affect. In input of *IPARM(36)* = 1 triggers an automatic recomputation of incomplete factorization or multilevel preconditioner with tighter thresholds (so as to compute a more accurate but expensive preconditioner) if the estimated norm of the error if more than 10 times higher than the limit in *DPARM(6)* after *IPARM(6)* iterations. The default value of *IPARM(36)* is 1.

- **IPARM(37:63), type R:**

  These are reserved for future use.

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

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

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

  If dynamic memory allocation by *WISMP* 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 Notes 2.2 and 2.4 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. An output value of -700 for *IPARM(64)* indicates an internal error and should be reported to *wsmp@watson.ibm.com*.

  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 less than or equal to *DPARM(11)* for incomplete Cholesky factorization or less than or equal to *DPARM(11)* in magnitude for incomplete LU factorization. If C-style (0-based) indexing is used and *IPARM(64)* > 0, then *IPARM(64)* is 1 + the index of the bad pivot.

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

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

The entries *DPARM(37)* through *DPARM(63)* are reserved. Unlike *IPARM*, only a few of the first 36 entries of *DPARM* are used. The description of only the relevant entries of *DPARM* is given below. Note that all reserved entries; i.e., *DPARM(37:63)* must contain 0.0.

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

  If an incomplete factorization of multilevel preconditioner is used, then at the end of the preconditioner computing phase, *DPARM(4)* contains the absolute value of the diagonal entry with the largest magnitude of the incomplete factor.

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

  If an incomplete factorization of multilevel preconditioner is used, then at the end of the preconditioner computing phase, *DPARM(4)* contains the absolute value of the diagonal entry with the smallest magnitude of the incomplete factor.

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

  The input in *DPARM(6)* is used as the target relative norm of the residual. The GMRES solver terminates when either number of iterations reaches *IPARM(6)*, or when the relative norm of the residual becomes smaller than *DPARM(6)*. Please refer to the description of *DPARM(8)* for the details on the exact stopping criteria of the conjugate gradient solver.

  The default value of *DPARM(6)* is 10^{-8}.

  The actual relative residual norm at the termination of iterations is returned in *DPARM(26)*.

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

  DPARM(7) is not used for the GMRES solver. For the conjugate gradient solver, *WISMP* computes an estimate of the Euclidean norm of the error in the solution [1]. The input in *DPARM(7)* is used as the target ratio of the estimated Euclidean norm of error to the norm of the computed solution. Please refer to the description of *DPARM(8)* for the details on the exact stopping criteria.

  The default value of *DPARM(6)* is 10^{-5}.

  If *WISMP* is reliably able to compute it, then the actual estimated relative Euclidean norm of the error at the termination of iterations is returned in *DPARM(27)*.

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

  DPARM(8) is not used for the GMRES solver. For the conjugate gradient solver, *DPARM(8)* is used to decide the termination criteria. If *DPARM(8)* is 0.0, which is the default, then the iterations terminate when the relative norm of the residual becomes smaller than *DPARM(6) and* the estimated relative Euclidean norm of the error becomes smaller than *DPARM(7)*. If *DPARM(8)* is 1.0, then the iterations terminate when *either* the relative norm of the residual becomes smaller than *DPARM(6)*, *or* the estimated relative Euclidean norm of the error becomes smaller than *DPARM(7)*. Of course, in either case, the number of iterations never exceed the limit specified in *IPARM(6)*).

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

  DPARM(11) is used and pivoting is opted for during the computation of a preconditioner based on incomplete LU factorization. This is the lower threshold on the value of a good diagonal entry. If a pivot value is less than or equal to *DPARM(11)*, then either an error is flagged or corrective action is taken based on the input in *IPARM(11)*. *DPARM(11)* must be non-negative.

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

  See the description of *IPARM(12)* for more details. *DPARM(12)* must be non-negative.

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

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

- **DPARM(14), type M:**

  The input *DPARM(14)* contains the threshold τ for incomplete factorization (used only if *IPARM(15)* is greater than or equal to 3). Please refer to the description of *IPARM(15)* for more details.

  If *DPARM(14)* is 0.0 on input, then *WISMP* selects its own value of τ based on its analysis of the input matrix and places is in *DPARM(14)*. The default input value of *DPARM(14)* is 0.0. Please refer to the description of *IPARM(34)* for some important details on the behavior of the solver when *DPARM(14)* is 0.0.

  On output, *DPARM(14)* may have a value different from the input value of *DPARM(14)*. The new value is the suggested value of *DPARM(14)*, which may result in a reduction of the total time spent in Tasks 3 and 4. Since *DPARM(14)* may be modified on output, its value needs to be reset for subsequent preconditioner generations if the user does not want to use the value suggested by *WISMP*.

- **DPARM(15), type M:**

  The input *DPARM(15)* contains the threshold γ for incomplete factorization (used only if *IPARM(15)* is greater than or equal to 3). Please refer to the description of *IPARM(15)* for more details.

  If *DPARM(15)* is 0.0 on input, then *WISMP* selects its own value of γ based on its analysis of the input matrix and places is in *DPARM(15)*. The default input value of *DPARM(15)* is 0.0. Please refer to the description of *IPARM(34)* for some important details on the behavior of the solver when *DPARM(15)* is 0.0.

  On output, *DPARM(15)* may have a value different from the input value of *DPARM(15)*. The new value is the suggested value of *DPARM(15)*, which may result in a reduction of the total time spent in Tasks 3 and 4. Since *DPARM(15)* may be modified on output, its value needs to be reset for subsequent preconditioner generation if the user does not want to use the value suggested by *WISMP*.

- **DPARM(21), 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.

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

  DPARM(22) is used to perturb small diagonal entries when row-column interchanges are not performed and the perturbation option is turned on by the user by setting *IPARM(12)* = 1. Please refer to the description of *IPARM(12)* for more details. *DPARM(22)* must be non-negative.

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

  This contains the number of floating point operations required for incomplete factorization.

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

  Please refer to the description of *IPARM(24)*. Along with *IPARM(24)*, *DPARM(24)* determines the compression efficiency while generating a supernodal matrix from the original coefficient matrix.

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

  After the solution step (Task 4), *DPARM(26)* contains the relative norm of the residual at the termination of the iterations.

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

  When the conjugate gradient solver is used, the the relative estimated norm of the error upon the termination of the iterations is returned in *DPARM(27)*. *DPARM(27)* is not used in the GMRES solver. A return value of 0.0 in *DPARM(27)* indicates that *WISMP* was not able to estimate the error norm.

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

  If the preconditioner type in *IPARM(15)* is 4 (multilevel preconditioner), then the input in *DPARM(28)* gives a termination criterion for the inner iterations to be performed at any level. Please refer to the description of *IPARM(28)* for more details.

4 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.

Note 4.1 *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 calling one of these routines. Placing an explicit undercore at the end of the routine name usually fixes the problem. For example, if calling* WS_SORTINDICES_I *does not work, then try calling* WS_SORTINDICES_I_.

4.1 *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 3.4. The description of *INFO* is similar to that of *IPARM(64)*.

Please read Note 4.1 at the beginning of this section.

4.2 *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 3.4. The description of *INFO* is similar to that of *IPARM(64)*.

Please read Note 4.1 at the beginning of this section.

4.3 *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 4.1 at the beginning of this section.

4.4 *WSETMAXTHRDS (MAXTHRD)*

If the underlying hardware is an SMP platform, then *WSMP* automatically spawns threads to utilize all the available CPU's. The number of threads spawned is a function of the the number of CPU's on the given node or workstation. The routine *WSETMAXTHRDS* can be used to set the number of threads that are spawned to correspond to *MAXTHRD* CPUs. For example, if you are running two processes on the same node that has four CPU's, you may want to set the maximum number of threads that each process uses to 2 in order to reduce the overheads. If you do not set the maximum number of threads to 2 in this scenario, then several (many more than four) threads will be competing for only 4 CPU's.

WSETMAXTHRDS is also useful when working on large SMP machines, where you would want to limit the number of CPUs that you want to use. For example, on a 64-CPU machine, *WSMP* will try to use all the CPUs by default which will result in excessive overhead.

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 4.10). Once *WSMP*/*PWSMP* is initialized, the number of threads cannot be changed for a given run.

If *WSETMAXTHRDS* is called with *MAXTHRD* = -1, then *WSMP* tries to use as many CPUs as are available in the hardware, which is the default mode.

4.5 *WSSYSTEMSCOPE* and *WSPROCESSSCOPE*

The default contention scope of the threads on AIX, Linux, SunOS, and Tru64 is *PTHREAD_SCOPE_SYSTEM* and is *PTHREAD_SCOPE_BOUND_NP* on IRIX. *WSSYSTEMSCOPE* can be called to change the contention scope to *PTHREAD_SCOPE_SYSTEM* and *WSPROCESSSCOPE* can be called to change the contention scope 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 4.10).

4.6 *WSETMAXSTACK (FSTK)*

All threads spawned by *WSMP* are, by default, assigned a 1 Mbyte stack in 32-bit mode (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 4 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 4.10). 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.

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.

4.7 *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 4.10).

4.8 *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*.

4.9 *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 4.1 at the beginning of this section.

4.10 *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 4.4), *WSSYSTEMSCOPE*, *WSPROCESSSCOPE* (Section 4.5), *WSETMAXSTACK* (Section 4.6), *WSETLF* (Section 4.7), and *WSETNOBIGMAL* (Section 4.8) 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 4.1 at the beginning of this section.

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

Both the serial and the parallel versions of the solver have the context stored internally, which enables them to perform a desired task at any time while using the information from tasks performed earlier, provided that the necessary information was generated at least once. For example, several calls to matrix-vector multiplication or iterative solution can be made with different numerical data (but the same indices) after one step of structural analysis. The solvers are able to perform these operations because they remember the results of the last structural analysis. Similarly, they remember the preconditioner for any number of iterative solves steps until a new preconditioner is generated or a new matrix structure is analyzed to replace the previously stored information. As a result, the solver routines occupy storage to remember all the information that might be needed for a future call to perform any legal task. The user can call a routine *WSMP_CLEAR()* in the serial/multithreaded mode and *PWSMP_CLEAR()* in the message-passing parallel mode to free this storage if required. After a call to any of these routines, the solver does not remember any context and the next call must be for performing a structural analysis (Task 1) to start a new context.

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

Please read Note 4.1 at the beginning of this section.

4.12 *WISFREE ()*^{S,T} and *PWISFREE ()*^{P}

WISFREE and *PWISFREE* release all the memory allocated by the iterative solver at the time of the call. If you need to solve more systems iteratively after a call to *WISFREE* or *PWISFREE*, you must start with analyzing the structure of the matrix (Task 1).

5 Support for Double Complex Data Type

The double complex (complex*16) version of the iterative solver can be accessed via routine *ZISMP*. This routine is identical to its double precision real counterpart with the exception that the data type of *AVALS*, *B*, *X*, and *RMISC* in this routine is *double complex* or *complex*16*. The *WSMP* web page at *http://www.cs.umn.edu/˜agupta/wsmp.html* contains example programs illustrating the use of this routine.

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

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

The HP-UX and some Linux versions of *WSMP* incorporates ATLAS BLAS (*http://math-atlas.sourceforge.net*). The following copyright notice, list of conditions, and the disclaimer applies to ATLAS.


```
*              Automatically Tuned Linear Algebra Software v3.6.0
*                     (C) Copyright 1998 R. Clint Whaley
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*   1. Redistributions of source code must retain the above copyright
*      notice, this list of conditions and the following disclaimer.
*   2. Redistributions in binary form must reproduce the above copyright
*      notice, this list of conditions, and the following disclaimer in the
*      documentation and/or other materials provided with the distribution.
*   3. The name of the ATLAS group or the names of its contributers may
*      not be used to endorse or promote products derived from this
*      software without specific written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
* ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
* TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE ATLAS GROUP OR ITS CONTRIBUTORS
* BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
```


7 Acknowledgements

We would like to thank Rogeli Grima, Mahesh Joshi, Felix Kwok, Chen Li, and Lexing Ying for their contributions to this project. The HP-UX and some Linux versions of the software incorporates ATLAS BLAS (*http://math-atlas.sourceforge.net*). We wish to thank the authors of ATLAS.

References

[1] Owe Axelsson and Igor Kaporin. Error norm estimation and stopping criteria in preconditioned conjugate gradient iterations. *Numerical Linear Algebra with Applications*, 8:265–286, 2001.

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

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

[5] Anshul Gupta. Graph partitioning based sparse matrix ordering algorithms for interior-point methods. Technical Report RC 20467 (90480), IBM T. J. Watson Research Center, Yorktown Heights, NY, May 21, 1996.

[6] Anshul Gupta. WSMP: Watson sparse matrix package (Part-II: direct solution of general sparse systems). Technical Report RC 21888 (98472), IBM T. J. Watson Research Center, Yorktown Heights, NY, November 20, 2000. *http://www.cs.umn.edu/~agupta/wsmp*.

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

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

[9] Ronald B. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995.

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