IBM Research Report

WSMP: Watson Sparse Matrix Package
Part III – iterative solution of sparse systems
Version 17.07

Anshul Gupta
IBM T. J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598
anshul@us.ibm.com

Haim Avron
IBM T. J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598
haimav@us.ibm.com
WSMP: Watson Sparse Matrix Package
Part III – iterative solution of sparse systems
Version 17.07

Anshul Gupta and Haim Avron
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
anshul@us.ibm.com and haimav@us.ibm.com

IBM Research Report RC 24398 (W0711-017)
November 5, 2007
©IBM Corporation 1997, 2017. All Rights Reserved.
Contents

1 Introduction to Part III 4

2 Recent Changes 4

3 Obtaining, Linking, and Running WSMP 4

3.1 AIX on IBM pSeries platform 5
3.1.1 Linking the libraries 5
3.1.2 Setting environment variables and running 6
3.1.3 Linking and execution-time problems 7
3.2 Linux on x86_64 platforms 7
3.2.1 Cygwin on Windows 7 8
3.3 Other platforms 8
3.3.1 Cray XE6 8
3.3.2 Mac OS 8
3.4 Controlling the number of threads 8
3.5 The number of MPI ranks per shared-memory unit 9

4 Description of Functionality 9

4.1 Types of matrices accepted and their input format 11
4.2 Krylov subspace solvers supported 11
4.3 Preconditioners and their parameters 11
4.4 Calling sequence of the WISMP subroutine 11

4.4.1 N (type I): matrix dimension 13
4.4.2 IA (type I): row/column pointers 13
4.4.3 JA (type I or M): column/row indices 13
4.4.4 AVALS (type I or M): nonzero values of the coefficient matrix 13
4.4.5 B (type I or O): right-hand side vector/matrix 14
4.4.6 LDB (type I): leading dimension of B 14
4.4.7 X (type O or I): solution vector/matrix 14
4.4.8 LDX (type I): leading dimension of X 14
4.4.9 NRHS (type I): number of right-hand sides 14
4.4.10 RMISC (type I, O, M): double precision output info 15
4.4.11 CVGH (type O): double precision convergence history output 15
4.4.12 IPARM (type I, O, M, and R): integer array of parameters 15
4.4.13 DPARM (type I, O, M, and R): double precision parameter array 25

5 Miscellaneous Routines 28

5.1 WS_SORTINDICES_I (M, N, IA, JA, INFO)$^S,T$ 28
5.2 WS_SORTINDICES_D (M, N, IA, JA, AVALS, INFO)$^S,T$ 28
5.3 WS_SORTINDICES_Z (M, N, IA, JA, AVALS, INFO)$^S,T$ 28
5.4 WSETMAXTHRDS (NUMTHRDS) 28
5.5 WSYSYSTEMSCOPE and WSPROCESSSCOPE 29
5.6 WSETMAXSTACK (FSIK) 29
5.7 WSETLF (DLF)$^T,P$ 29
5.8 WSETNOBIGMAL() 29
5.9 WSMP_VERSION (V, R, M) 30
5.10 WSMP_INITIALIZE()$^S,T$ and PW SMP_INITIALIZE()$^P$ 30
5.11 WSMP_CLEAR()$^S,T$ and PW SMP_CLEAR()$^P$ 30
5.12 WISFREE()$^S,T$ and PWISFREE()$^P$ 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. 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 Part III, the iterative solution of sparse systems of linear equations in WSMP. Parts I and II of User’s Guide [5] can be obtained from http://www.research.ibm.com/projects/wsmp, along with some example programs and technical papers related to the software. A current list of known bugs and issues is also maintained at this web site. Unlike the direct solvers, the iterative solvers described in this document are available in a shared-memory parallel environment only. Please disregard any portions of this document that may refer to distributed-memory/message-passing parallelism.

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

The organization of this document is as follows. Section 2 describes important recent changes in the software that may affect the users of earlier versions. Section 3 lists the various libraries that are available and describe how to obtain and use the libraries. Section 4 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 5 describes a few utility routines available to the users. Section 6 gives a brief description of the double-complex data type interface of WSMP’s iterative solvers. Section 7 contains the terms and conditions that all users of the package must adhere to.

2 Recent Changes

The role and interpretation of IPARM(12) has changed in Version 14.6.

IPARM(34) (iparm[33]) to choose symmetry enhancing dropping has been introduced in Version 14.5.

The default values of DPARM(6) and DPARM(7) (dparm[5] and dparm[6]) have changed in Version 14.3.

3 Obtaining, Linking, and Running WSMP

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

The software also needs a small scratch space on then disk and uses the /tmp directory for that. You can override the default by setting the environment variable TMPDIR to another location. \[1\]

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

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

Note 3.1 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.

---

\[1\]This is particularly useful on Cray XE6 compute nodes where /tmp may be unavailable.
WSMP performs its own parallelization and expects all its BLAS calls to run on a single thread. BLAS calls running in parallel can cause substantial performance degradation. With some BLAS libraries, it may be necessary to set the environment variable OMP_NUM_THREADS to 1. Many BLAS libraries have their own environment variable, such as MKL_NUM_THREADS or GOTO_NUM_THREADS, which should be used if available.

Note 3.2 On many systems, the user may need to increase the default limits on stack size and data size. Failure to do so may result in a hung program or a segmentation fault due to small stack size and a segmentation fault or an error code (IPARM(64)) of −102 due to small size of the data segment. Often the limit command can be used to increase stacksize and datasize. When the limit command is not available, please refer to the related documentation for your specific system. Some systems have separate hard and soft limits. Sometimes, changing the limits can be tricky and can require root privileges. You may download the program memchk.c from www.research.ibm.com/projects/wsmp and compile and run it as instructed at the top of the file to see how much stack and data space is available to you.

3.1 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. The ESSL library (or another BLAS library optimized for IBM pSeries hardware) – see Note 3.1

3.1.1 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, and -lm_r provides the reentrant math functions library.

1. 64-bit serial/SMP mode:
   xlf_r <source/object files> -o <executable target> -q64 -b64 -lpthread -lhm -lhu -lm_r -lessl -lwsmpe64 -L<path to parent directory of libwsmp64.a> -bdatapsize:64k

2. In 64-bit message-passing mode:
   mpxlf_r <source/object files> -o <executable target> -q64 -b64 -lhm -lhu -lm_r -lpthread -lessl -lpwsmpe -L<path to parent directory of libpwsmp64.a> -bdatapsize:64k
   mpcc_r <source/object files> -o <executable target> -q64 -b64 -lxlf90_r -lhm -lhu -lm_r -lpthread -lessl -lpwsmpe -L<path to parent directory of libpwsmp64.a> -bdatapsize:64k

Note 3.3 The use of -lhm -lhu while linking increases the speed of dynamic memory allocation and usually improves the performance of the solver.

Note 3.4 In the 64-bit mode, it is a good idea to use -bmaxdata:<membytes> while linking, where membytes is the amount of physical memory on the machine (or each node in the message-passing mode). This will prevent the solver from paging. Note that you may still experience paging if there are other applications running on your machine.
Note 3.5 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 3.6 Note that the XLF Fortran library must be linked by using -lxlf90 if the xlc command is used for creating the executable. The math library should also be linked in by adding -lm to the linking command as shown above.

### 3.1.2 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 cores. 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.

Additionally, 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, if smt_threads are disabled on your system, then you can set the environment variable RT_GRQ to ON to obtain better parallel speedups from multithreading. However, if smt_threads are enabled, then setting RT_GRQ to ON can actually degrade performance. If you do not know whether smt_threads are enabled on your system or not, then you can try running your application with RT_GRQ=ON and RT_GRQ=OFF, and chose the setting that works better.

Setting the environment variable LDR_CNTRL to DATAPSIZE=64K can give a significant performance boost if the default page size on your system is 4KB. Alternately, you can use -bdatapsize:64k while linking to create your executable.

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
  export MP_EUILIB=us
  export MP_EUIDEVICE=css0
  export MP_CSS_INTERRUPT=yes
  export MP_WAIT_MODE=nopoll
  export MP_PULSE=0

  csh or tcsh
  setenv MP_EUILIB us
  setenv MP_EUIDEVICE css0
  setenv MP_CSS_INTERRUPT yes
  setenv MP_WAIT_MODE nopoll
  setenv MP_PULSE 0
```

Note that the WSMP and PWSMP libraries use Pthreads to exploit shared-memory parallelism on each process. It is usually best to use fewer processes on each node than the number of cores per node. If you are using multiple processes per compute node, please be aware that by default, each process will attempt to use all the cores on the node. Please refer to Section 5.4 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 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.
3.1.3 Linking and execution-time problems

Please refer to Section 3.1.1 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.

3.2 Linux on x86_64 platforms

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

The most important consideration while using the distributed-memory parallel versions of WSMP on a Linux platform is that MPI library may not be thread safe. The symmetric solver needs MPI_THREAD_FUNNELED support and the unsymmetric solver needs MPI_THREAD_MULTIPLE support. Therefore, MPI must be initialized accordingly. If MPI_THREAD_MULTIPLE support is not available, then you can use only one thread per MPI process. This can be accomplished by following the instructions in Section 5.4.

Note 3.7 With most MPI implementations, when using more than one thread per process, the user will need to initialize MPI using MPI_INIT_THREAD and request the appropriate level of thread support. The default level of thread support granted by using MPI_INIT may not be sufficient, particularly for the unsymmetric solver. You may also need to use the -mt mpi flag while linking with Intel MPI for the unsymmetric solver.

Note 3.8 There may be environment variables specific to each MPI implementation that need to be used for obtaining the best performance. Examples of these include MV2_ENABLE_AFFINITY with mvapich2 and I_MPI_PIN, I_MPI_PIN_MODE, I_MPI_PIN_DOMAIN etc. with Intel MPI.

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

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

In some cases, WSMP libraries that do not require a Fortran compiler and can be linked using gcc are available in the GCC subdirectory of the distribution.

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

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

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

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

An example of linking your program with the message-passing library libpwsmp64.a on a cluster with x86_64 nodes is as follows:
mpif90 -o <executable> <user source or object files> <BLAS library> -lpwsmp64 -L<path of libpwsmp64.a> -lpthread

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

3.2.1 Cygwin on Windows 7
A 32-bit library compiled and tested in the Cygwin environment running under Windows 7 is available. An example of linking in Cygwin is as follows (very similar to what one would do on 32-bit Linux on an IA32 platform):

gfortran -o <executable> <user source or object files> -L<path of libwsmp64.a> -lwsmp -lblas -lpthread -lm -m32

3.3 Other platforms
WSMP is currently supported on the following platforms in addition to the ones listed previously in this document. If you need the WSMP or PWSMP libraries for any other platform and can provide us an account on a machine with the target architecture and operating system, we may be able to compile the libraries for you. Please send e-mail to wsmp@us.ibm.com to discuss this possibility.

3.3.1 Cray XE6
The Cray XE6 libraries are built with the PGI programming environment, and require the same for linking and running. It may be necessary to unload any other programming environment and load the PGI programming environment. An example of linking on Cray XE6 is as follows.

module load PrgEnv-pgi
ftn -o <executable> <user source or object files> -lpwsmp64 -lpthread -lacml

Please refer to Note 3.1. The environment variable OMP_NUM_THREADS must be set to 1 in order for each call to an ACML library function to use a single thread.

3.3.2 Mac OS
MAC OS libraries are available for Intel and GNU compilers. The BLAS can be provided by either explicitly linking MKL (preferred) or by using the Accelerate framework. Linking examples are as follows:
gfortran -o <executable> <user source or object files> -m32 -lwsmp -L<path of libwsmp.a> -lm -lpthread -framework Accelerate
gfortran -o <executable> <user source or object files> -m64 -lwsmp64 -L<path of libwsmp64.a> -lm -lpthread -framework Accelerate

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

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

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

WSMP provides two ways of controlling the number of threads that it uses. You can either use the function WSETMAXTHRDS (NUMTHRDS) described in Section 5.4 inside your program, or you can set the environment variable WSMP_NUM_THREADS to NUMTHRDS. If both WSETMAXTHRDS and the environment variable WSMP_NUM_THREADS are used, then the environment variable overrides the value set by the routine WSETMAXTHRDS.

### 3.5 The number of MPI ranks per shared-memory unit

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

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

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

### 4 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 preconditioned Krylov subspace methods (CG, GMRES, TFQMR, BiCGStab), for performing sparse matrix-vector multiplication, and for generating and solving with respect to incomplete factorization based preconditioners.

All WSMP routines can be called from Fortran as well as C or C++ programs using a single interface described in this document. As a matter of convention, symbols (function and variable names) are in capital letters in context of Fortran and in small letters in context of C. Please refer to Notes 4.1, 4.2, and 5.1 for more details on using WSSMP with Fortran or C programs.

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

In calls to WISMP, IPARM(2) and IPARM(3) control the subset of the tasks to be performed (see Section 4.4 for more details).

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

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

pattern but different values. Depending on the application, either a single symbolic analysis step may suffice for 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.

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, $n \geq 1, m \geq 1$. Currently, CG, GMRES, TFQMR, or BiCGStab algorithms can be used for obtaining the solution.

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

6. Task 6, Preconditioning Step: This task can be used to compute $X = M^{-1}B$, 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 4.4). 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 4.4.12 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 time savings in many applications. By an appropriate choice of IPARM(15), DPARM(14), and DPARM(15), Task 3 can be used to generate an exact direct factorization and Task 4 can be 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.

4.1 Types of matrices accepted and their input format

WISMP and ZISMP (Section 6) accept the sparse coefficient matrices in the compressed sparse row (CSR) or compressed sparse column (CSC) formats. Both symmetric and unsymmetric matrices of various types are accepted; please refer to Table 5 and the description of IPARM(7) in Section 4.4.12 for more details. Table 4 under the description of IPARM(4) shows how to indicate the choice of input format to WISMP or ZISMP.

For symmetric matrices, the user has the option of either providing 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 (IPARM(5) in Section 4.4.12), all data structures must follow the same numbering convention which must stay consistent through all the calls referring to a given system of equations.

4.2 Krylov subspace solvers supported

WISMP currently includes implementations of preconditioned conjugate gradient (CG) method, GMRES, TFQMR, and BiCGStab. WISMP’s current focus is on robust high-performance preconditioners. As described in the beginning of Section 4, 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.

4.3 Preconditioners and their parameters

WISMP currently supports Jacobi (diagonal), Gauss-Siedel (SSOR with relaxation $ω = 1$), and Incomplete Cholesky/LDL$^T$ 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. The preconditioner type is chosen by means of IPARM(15).

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(28), 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.

4.4 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 $\bar{X}$ is computed by WISMP.
The storage of this matrix in the A 9 X 9 symmetric sparse matrix.

The input formats accepted by WISMP is shown in the table.

A 9 X 9 symmetric sparse matrix.

Figure 1: Illustration of the upper triangular CSR and full CSR formats for a symmetric matrix for the serial/multithreaded WISMP routine.
Note 4.1  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 interpret it accordingly. For example, IPARM(11) will actually be IPARM[10] in a C program calling WISMP.

Note 4.2  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 4.4 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 )

        wismp_ ( int *n, int ia[], int ja[], double avals[], double b[], int *ldb, double x[], int *ldx, int *nrhs, double rmisc[], double cvgh[], int iparm[], double dparm[] )

4.4.1  N (type I): matrix dimension

        INTEGER N
        int *n

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

4.4.2  IA (type I): row/column pointers

        INTEGER IA (N + 1)
        int ia[]

        IA is an integer array of size one greater than N. IA(I) points to the first row/column index of column/row I in the array JA. Refer to Figure 1 and description of IPARM(4) in Section 4.4.12 for more details.

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

        INTEGER JA ( * )
        int ja[]

        The integer array JA contains the column (row) indices of the upper (lower) triangular part of the symmetric sparse matrix A. The indices of a column (row) are stored in consecutive locations. In addition, these consecutive column (row) indices of a row (column) must be sorted in increasing order upon input. WSMP provides two utility routines to sort the indices (see Section 5 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.

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

        DOUBLE PRECISION AVALS ( * )
        double avals[]

        The array AVALS contains the actual double precision values corresponding to the indices in JA. The size of AVALS is the same as that of JA. See Figure 1 for more details. Please note that, in many cases, AVALS is accessed during Task 1 (see description of IPARM(2..3) in Section 4.4.12), 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 4.3 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.

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

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

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

4.4.6 LDB (type I): leading dimension of B

```c
INTEGER LDB
int *ldb
```

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

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

```c
DOUBLE PRECISION X ( LDX, NRHS )
double x[]
```

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.

4.4.8 LDX (type I): leading dimension of X

```c
INTEGER LDX
int *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.

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

```c
INTEGER NRHS
int *nrhs
```

\( NRHS \) is the second dimension of \( B \) and \( X \); it is the number of right-hand sides that need to be solved for. It must be a nonnegative integer.
4.4.10 RMISC (type I, O, M): double precision output info

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

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 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.

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

```c
DOUBLE PRECISION CVGH ( 0 : IPARM(6) )
double cvgh[]
```

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.

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

```c
INTEGER IPARM ( 64 )
int 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) or iparm[0], type I or M:**

  If IPARM(1) is 0, then the remainder of the IPARM array and the DPARM array are filled with default values by 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) or iparm[1], type M:**

  As described in Section 4, 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) ≤ 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.
### 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).

<table>
<thead>
<tr>
<th>Index</th>
<th>Default</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mandatory I/P</td>
<td>default/user defined</td>
<td>M</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>mandatory I/P</td>
<td>starting task</td>
<td>M</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>mandatory I/P</td>
<td>last task</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>I/P format</td>
<td>I</td>
<td>-</td>
<td>max. fact. diag.</td>
<td>O</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>numbering style</td>
<td>I</td>
<td>-</td>
<td>min. fact. diag.</td>
<td>O</td>
</tr>
<tr>
<td>6</td>
<td>1000</td>
<td>max iterations</td>
<td>I</td>
<td>$1 \times 10^{-7}$</td>
<td>target rel. resid. norm</td>
<td>I</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>matrix type</td>
<td>I</td>
<td>$1 \times 10^{-4}$</td>
<td>target rel. error norm</td>
<td>I</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>max. matching use</td>
<td>I</td>
<td>0.0</td>
<td>termination criterion</td>
<td>I</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>max. matching frequency</td>
<td>I</td>
<td>0.0</td>
<td>diagonal perturbation</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>scaling option</td>
<td>I</td>
<td>$10^{-18}$</td>
<td>singularity threshold</td>
<td>I</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>thresh. pivoting opt.</td>
<td>I</td>
<td>$1 \times 10^{-3}$</td>
<td>pivot thresh.</td>
<td>I</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>no. of attempts</td>
<td>I</td>
<td>$10^{-5}$</td>
<td>small piv. thresh.</td>
<td>I</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>solver choice</td>
<td>I</td>
<td>-</td>
<td>no. of supernodes</td>
<td>O</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>AVALS/JA reuse opt.</td>
<td>I</td>
<td>0.0</td>
<td>threshold $\tau$</td>
<td>M</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>preconditioner choice</td>
<td>I</td>
<td>0.0</td>
<td>threshold $\gamma$</td>
<td>M</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>ordering option 1</td>
<td>O</td>
<td>-</td>
<td>number of zero diagonals</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>ordering option 2</td>
<td>O</td>
<td>-</td>
<td>number of missing/-ve diagonals</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>max imbalance</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>refinement strategy</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>matrix characteristics</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>GMRES restart</td>
<td>I</td>
<td>-</td>
<td>structural symmetry</td>
<td>O</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td># approx. eigenvecs</td>
<td>I</td>
<td>$10^{-5}$</td>
<td>small piv. repl.</td>
<td>I</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>incomplete fact. size</td>
<td>O</td>
<td>-</td>
<td>incomplete fact. ops.</td>
<td>O</td>
</tr>
<tr>
<td>24</td>
<td>40</td>
<td>max. clique size</td>
<td>I</td>
<td>0.8</td>
<td>min. compression efficiency</td>
<td>I</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>RMISC use</td>
<td>I</td>
<td>1.0</td>
<td>direct solver drop tolerance factor</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>-</td>
<td>number of iterations</td>
<td>O</td>
<td>-</td>
<td>relative residual norm</td>
<td>O</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>CVGH use</td>
<td>I</td>
<td>-</td>
<td>error norm estimate</td>
<td>O</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>$\tau$, $\gamma$ tuning</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>mat-vec tuning</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td># diags replaced/interchanges</td>
<td>O</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>31</td>
<td>-</td>
<td># entries dropped</td>
<td>O</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>precondition repeat use</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>33</td>
<td>-</td>
<td>no. of CPU’s used</td>
<td>O</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>symm. enh. dropping</td>
<td>I</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
<tr>
<td>35-63</td>
<td>0</td>
<td>reserved</td>
<td>R</td>
<td>0.0</td>
<td>reserved</td>
<td>R</td>
</tr>
<tr>
<td>64</td>
<td>-</td>
<td>return error code</td>
<td>O</td>
<td>-</td>
<td>unused</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>IPARM(2)</th>
<th>IPARM(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 1, 2, 3, 4</td>
</tr>
<tr>
<td>1</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>2</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>3</td>
<td>3, 4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
On output, \texttt{IPARM(2)} contains $1 + \text{number of the last task performed by WISMP}$, if any. This is to facilitate users to restart processing on a problem from where the last call to \texttt{WISMP} left it. Also, if \texttt{WISMP} is called to perform multiple tasks in the same call and it returns with an error code in \texttt{IPARM(64)}, then the output in \texttt{IPARM(2)} indicates the task that failed. If \texttt{WISMP} performs no task, then, on output, \texttt{IPARM(2)} is set to $\text{max(IPARM(2),IPARM(3)+1)}$.

When using \texttt{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 \texttt{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 $\text{NRHS}$ 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 will be used, but may still result 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. \texttt{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 \texttt{WSMP}'s direct solvers to obtain fast solutions to a sequence of linear systems with gradually varying coefficient matrices. As described in Section 4.4 by choosing appropriate values of \texttt{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 \texttt{WISMP} can automatically detect where, if at all, it needs to recompute the preconditioner within a repeated sequence of Tasks 2 and 4.

\texttt{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 \texttt{WISMP} to provide these functions for their own iterative algorithms other than CG, GMRES, BiCGStab, or TFQMR, currently provided by \texttt{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 \texttt{WISMP} to perform Tasks 5 and 6 if the user chooser to implement their own iterative method and use \texttt{WISMP}'s fast parallel sparse matrix-vector multiplication and preconditioning algorithms.

Please refer to Note 4.4.12 for some important information related to Task 3.

- \texttt{IPARM(3)} or \texttt{iparm[2]}, type I:

\texttt{IPARM(3)} must contain the number of the last task to be performed by \texttt{WISMP}. In a call to \texttt{WISMP}, all tasks from \texttt{IPARM(2)} to \texttt{IPARM(3)} are performed (both inclusive). If \texttt{IPARM(2)} > \texttt{IPARM(3)} or both \texttt{IPARM(2)} and \texttt{IPARM(3)} is out of the range 1–6, then no task is performed. This can be used to fill \texttt{IPARM} and \texttt{DPARM} with default values; e.g., by calling \texttt{WISMP} with \texttt{IPARM(1)} = 0, \texttt{IPARM(2)} = 0, and \texttt{IPARM(3)} = 0.

- \texttt{IPARM(4)} or \texttt{iparm[3]}, type I:

\texttt{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 \texttt{IPARM(4)}.

The default value of \texttt{IPARM(4)} is 1.

- \texttt{IPARM(5)} or \texttt{iparm[4]}, type I:
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).

| 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).

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

  IPARM(7) denotes 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)** or iparm[7], 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 rows such that the product of the absolute values of the diagonal is maximized [9][1] [7][6]. 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)** or iparm[8], 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
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) or iparm[9], 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) or iparm[10], type I:**
  IPARM(11) is ignored for matrix types 1, 3, and 4 (Table 4.4.12 describes matrix types).
  For matrix types 0 and 2, if a preconditioner based on incomplete factorization is chosen, then IPARM(11) determines whether or not partial pivoting is performed during incomplete LU factorization. If IPARM(11) = 0, then partial pivoting is turned off. If IPARM(11) = 1, then partial pivoting is turned on. If IPARM(11) = 2, which is also the default, then WISMP decides whether or not to use partial pivoting based on the characteristics of the matrix.
  The total number of row or column interchanges performed are reported in IPARM(30).
  Although IPARM(11) pertains to the preconditioner generation phase (Task 3) when incomplete factorization is used, its value must be set before starting Task 1 because certain data structures that are set up during the structural analysis phase (Task 1) must account for pivoting, if needed.

- **IPARM(12) or iparm[11], type I:**
  IPARM(12) is ignored for the values of preconditioner type (IPARM(15)) less than 3. Please refer to the description of IPARM(14), whose default nonzero setting makes IPARM(12) ineffective. If IPARM(14) is set to 0 by the user, then IPARM(12) has the following effect for incomplete factorization preconditioning.
  When IPARM(12) is effective and incomplete factorization preconditioning is chosen, then WISMP may make up to \(k\) attempts at computing the incomplete factorization, where \(k\) is the absolute value of the input in IPARM(12). In each of the \(k - 1\) reattempts if the first attempt fails, a preconditioner with progressively tighter thresholds (i.e., a more accurate but expensive preconditioner) is computed. An incomplete factorization is deemed to have failed if the estimated norm of the error is more than 10 times higher than the limit in DPARM(6) after IPARM(6) iterations in the previous attempt.
  If IPARM(12) is positive, then WISMP may attempt a complete factorization, depending on the results of the previous incomplete factorization attempts. If IPARM(12) is negative, then complete factorization is not considered. For example, with a 3 or a -3 in IPARM(12), WISMP will make at most 3 attempts to solve the system—one using the incomplete factorization based on the original values of \(\tau\) and \(\gamma\) (DPARM(14..15)) and at most two more with progressively denser preconditioners. However, with IPARM(12) = -3, the factorization will be incomplete in all attempts, whereas WISMP may decide to perform a complete factorization if IPARM(12) = 3. IPARM(12) = 0 is treated as IPARM(12) = 1.
  The default value of IPARM(12) is 2. However, the default value of IPARM(14) is nonzero, which makes the IPARM(12) input ineffective. So if an automatic reattempt is desirable, then IPARM(14) must be changed to 0 from its default value.

- **IPARM(13) or iparm[12], type I:**
  If IPARM(13) is 0 (default), then 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 the GMRES method is used. BiCGStab or TFQMR can be selected by setting IPARM(13) to 3 or 4, respectively. Flexible GMRES (FGMRES) can be chosen by setting IPARM(13) to 6. Note that an input value of 5 is currently unused.
<table>
<thead>
<tr>
<th>Preconditioner type</th>
<th>IPARM(15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No preconditioning</td>
<td>0</td>
</tr>
<tr>
<td>Diagonal preconditioner</td>
<td>1</td>
</tr>
<tr>
<td>SSOR preconditioner</td>
<td>2</td>
</tr>
<tr>
<td>Incomplete factorization</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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

  IPARM(14) can be used to reuse the space in the arrays AVALS and JA during the preconditioner generation for preconditioner type 3 (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.

  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). In this case, the modified AVALS contains preconditioner data, must be passed unaltered to the solve phase (Task 4) after preconditioner generation (Task 3).

  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. It should also be set to 0 for IPARM(12) to be effective.

- **IPARM(15) or iparm[14], 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.

  An incomplete factorization based on two thresholds, τ and γ is used for preconditioner type 3. 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) and DPARM(15) 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 4.4 Even if IPARM(15) is 0; i.e., no preconditioning is used, Task 3 must still 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) or iparm[15], type I:

  WISMP uses a preordering of the matrix rows and columns to minimize fill during incomplete factorization for preconditioner type. 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 [2] is performed. If IPARM(16) is 1, 2, or 3, then a graph-partitioning based ordering [3] 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 0, in which case, WISMP choses the best ordering automatically.

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

  If an incomplete factorization based preconditioner is used, then IPARM(17) can be used to choose the partitioning and ordering heuristics that WISMP applies to 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.

  Four other combinations are available and can be selected by setting IPARM(17) to 1, 2, 3, or 4, respectively. In addition to the default, users are encouraged to experiment with the other four as well to determine which one works best for their problems.

- IPARM(18) or iparm[17], type I:

  WISMP uses multilevel graph partitioning algorithms [4] 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) or iparm[18], 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 [4] for details on refinement strategies.

  The default value of IPARM(19) is 0.

- IPARM(20) or iparm[19], 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 preconditioner type 3) may fail for some matrices with very irregular structure, unless \( IPARM(20) \) is set to 1.

- **IPARM(21) or iparm[20], type I:**
  \( IPARM(21) \) is read only when the GMRES solver is used. If 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) or iparm[21], type I:**
  \( IPARM(22) \) is read only when the GMRES solver is used. 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 [3]. 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) or iparm[22], type O:**
  If an incomplete factorization preconditioner is used, then after the preconditioner generation phase, \( IPARM(23) \) contains the number of thousands of double words required to store the incomplete factors.

- **IPARM(24) or iparm[23], 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) or iparm[24], type I:**
  Please refer to the description of RMISC in Section 4.4.10.

• **IPARM(26) or iparm[25], 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) or iparm[26], 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) or iparm[27], type I:**
  If both inputs DPARM(14) and DPARM(15) are 0.0, and the preconditioner type (IPARM(15)) is 3, 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 \( \tau \) and \( \gamma \) (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(28) 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(28) to 0 guarantees the same solution for the same problem each time. Other valid values of IPARM(28) are 1, 2, and 3. A higher value of IPARM(28) can potentially result in a higher degree of variation in results between different runs, but usually also an overall faster solution.

  The default value of IPARM(28) is 2.

• **IPARM(29) or iparm[28], 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(29) is 1, which enables this optimization. It can be switched off by setting IPARM(29) to 0, in which case, a fixed block size will be used.

• **IPARM(30) or iparm[29], type O:**
  When very small diagonal entries are replaced based on the input in DPARM(12) during incomplete Cholesky factorization, 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) or iparm[30], type O:**
  At the end of the preconditioner generation step, IPARM(31) contains the number of thousands of entries dropped during incomplete factorization preconditioner computation.

• **IPARM(32) or iparm[31], 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) or iparm[32], 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. Please refer to Section 3.4 for details on controlling the number of threads in WSMP.

• **IPARM(34) or iparm[33], type I:**
  The default value of IPARM(34) is 0 and has no effect. When IPARM(34) is set 1, the dropping strategy during incomplete factorization (for preconditioner type 3 in IPARM(15)) attempts to reduce structural unsymmetry in the factors. This strategy can yield improved convergence rates and/or smaller factors for some matrices that are structurally unsymmetric. The users should experiment with setting IPARM(34) to 0 and 1 to determine which option works best for their application if the coefficient matrix is structurally unsymmetric.

• **IPARM(35:63) or iparm[34:62], type R:**
  These are reserved for future use.

• **IPARM(64) or iparm[63], type O:**
  In the event of a successful return from 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 3.2 and 3.4 if you get this error in your program.
  An error code of $-300$ is returned if the current operation is invalid because it depends on the successful completion of another operation, which failed or was not performed by the user. For example, if the preconditioner generation fails and you call WSMP to perform a solve phase after the failed call for preconditioner generation, you can expect error $-300$.
  An output value of $-700$ for IPARM(64) indicates an internal error and should be reported to wsmp@us.ibm.com.
  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.
4.4.13 DPARM (type I, O, M, and R): double precision parameter array

DOUBLE PRECISION DPARM ( 64 )

double 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) or dparm[3], type O:**
  If an incomplete factorization based 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) or dparm[4], type O:**
  If an incomplete factorization based preconditioner is used, then at the end of the preconditioner computing phase, DPARM(5) contains the absolute value of the diagonal entry with the smallest magnitude of the incomplete factor.

- **DPARM(6) or dparm[5], type I:**
  The input in DPARM(6) is used as the target relative norm of the residual. The iterative 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 $10^{-7}$.
  The actual relative residual norm at the termination of iterations is returned in DPARM(26).
  Note that the quantities in DPARM(6) and DPARM(26) must be interpreted relative to norm of the residual with respect to the initial estimate to the solution. If IPARM(25) is even (0 or 2), then the initial residual norm is always equal to the norm of $B$ because WISMP chooses $X = 0$ as the initial guess of the solution. However, if IPARM(25) is odd (1 or 3) and the user supplies an initial estimate to the solution in RMISC, then the relative residual norm can be large even if the absolute norm is small, depending on the accuracy of the input in RMISC. Therefore, an accurate description of DPARM(6) would be that it is the inverse of the factor by which WISMP strives to reduce the norm of the residual.

- **DPARM(7) or dparm[6], type I:**
  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(7) is $10^{-4}$.
  The estimate of the relative error norm at the termination of iterations is returned in DPARM(27).
  Unlike the residual norms in DPARM(6) and DPARM(26), which are relative (to the initial residual) quantities, the error norm estimates in DPARM(7) and DPARM(27) are absolute quantities.

- **DPARM(8) or dparm[7], type I:**
  The input in DPARM(8) is used to decide the termination criterion. 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(9) or dparm[8], type I:**

  *DPARM(9)* can be used to minimize the chances of the failure of incomplete Cholesky factorization for symmetric/Hermitian positive definite matrices. Incomplete Cholesky factorization may fail due to a negative diagonal entries. *WSMP* boosts the diagonal dominance of the original coefficient matrix *A* by introducing a perturbation that adds *DPARM(9)* to the diagonal entries of *A*.

  If *DPARM(9)* is set to 0.0, which is its default value, then *WSMP* chooses an appropriate value of perturbation, only if needed. If *DPARM(9) > 0.0*, then the input value of *DPARM(9)* is used to perturb the matrix. If *DPARM(9) < 0.0*, then this correction is not applied and the preconditioner is permitted to become indefinite. For some problems, omitting this diagonal perturbation results in a better preconditioner. Note that even after increasing the magnitude of the diagonal by applying this perturbation, a negative entry may be encountered on it. If that happens, the preconditioner is permitted to become indefinite without further corrective action.

  If the matrix is equilibrated using *IPARM(10) = 1* (i.e., all diagonals are 1.0 prior to incomplete factorization) and you decide to supply a positive value in *DPARM(9)*, then something in the $10^{-2} - 10^{-3}$ range is usually a good value.

• **DPARM(10) or dparm[9], type I:**

  The input in *DPARM(10)* is relevant only for incomplete factorization preconditioners and is used as the threshold for determining if the preconditioner has become singular. If a leading row or column is encountered in the unfactored part of the matrix such that all its entries are less than or equal to *DPARM(10)*, then the incomplete factor is deemed singular and a correction is applied so that factorization can continue. The default value of *DPARM(10)* is $10^{-18}$. The default value of *DPARM(10)* is appropriate only if the matrix is scaled. If the matrix is not scaled, then the user must specify an appropriate threshold in *DPARM(10)* to detect singularity.

• **DPARM(11) or dparm[10], type I:**

  *DPARM(11)* is used when 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 a row/column interchange is performed to reduce growth in incomplete LU factorization. *DPARM(11)* must be non-negative. A value of 0.0 in *DPARM(11)* has the effect of turning pivoting off, even if *IPARM(11) is 1 or 2*. The default value of *DPARM(11)* is $10^{-3}$.

• **DPARM(12) or dparm[11], type M:**

  *DPARM(12)* is ignored for matrix types 0, 1, and 2 (Table 5). Let $\alpha = A(i, i)$ just before the $i$-th step of incomplete Cholesky 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 $\alpha$ and whose magnitude is $|A(l, i)| \times *DPARM(22)*$. For incomplete Cholesky factorization, the perturbed diagonal entry is always positive, irrespective of its pre-perturbation value. *DPARM(12)* must be non-negative. The default value of *DPARM(12)* is $10^{-5}$.

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

• **DPARM(13) or dparm[12], 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) or dparm[13], type M:**

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

On output, $\text{DPARM}(14)$ may have a value different from the input value of $\text{DPARM}(14)$. The new value is the suggested value of $\text{DPARM}(14)$, which may result in a reduction of the total time spent in Tasks 3 and 4. Since $\text{DPARM}(14)$ 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 $\text{WISMP}$.

- **$\text{DPARM}(15)$ or $\text{dparm}[14]$, type M:**
The input $\text{DPARM}(15)$ contains the threshold $\gamma$ for incomplete factorization (used only if $\text{IPARM}(15)$ is greater than or equal to 3). Please refer to the description of $\text{IPARM}(15)$ for more details.

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

  On output, $\text{DPARM}(15)$ may have a value different from the input value of $\text{DPARM}(15)$. The new value is the suggested value of $\text{DPARM}(15)$, which may result in a reduction of the total time spent in Tasks 3 and 4. Since $\text{DPARM}(15)$ 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 $\text{WISMP}$.

- **$\text{DPARM}(16)$ or $\text{dparm}[15]$, type O:**
  At the end of Task 1 (see Table[1] for description of tasks), $\text{DPARM}(16)$ returns the number of diagonal entries that are zero in the original matrix. This output is fragile for complex matrices. For positive definite and diagonally dominant matrices, it is assumed without checking that there are no zero-valued diagonal entries.

- **$\text{DPARM}(17)$ or $\text{dparm}[16]$, type O:**
  For symmetric indefinite matrices, at the end of Task 1 (see Table[1] for description of tasks), $\text{DPARM}(17)$ returns the number of diagonal entries that are negative in the original matrix. For general unsymmetric matrices, $\text{DPARM}(17)$ returns the number of missing diagonal entries. This output is fragile for complex matrices. For positive definite and diagonally dominant matrices, it is assumed without checking that there are no missing or negative diagonal entries.

- **$\text{DPARM}(21)$ or $\text{dparm}[20]$, type O:**
  $\text{DPARM}(21)$ 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.

- **$\text{DPARM}(22)$ or $\text{dparm}[21]$, type I:**
  Please refer to the description of $\text{DPARM}(12)$ for more details. $\text{DPARM}(22)$ must be non-negative. The default value of $\text{DPARM}(22)$ is $10^{-5}$.

- **$\text{DPARM}(23)$ or $\text{dparm}[22]$, type O:**
  This contains the number of floating point operations required for incomplete factorization.

- **$\text{DPARM}(24)$ or $\text{dparm}[23]$, type I:**
  Please refer to the description of $\text{IPARM}(24)$. Along with $\text{IPARM}(24)$, $\text{DPARM}(24)$ determines the compression efficiency while generating a supernodal matrix from the original coefficient matrix.

- **$\text{DPARM}(25)$ or $\text{dparm}[24]$, type I:**
  $\text{WISMP}$ partitions the matrix into as many domains as the number of threads. Each thread performs incomplete factorization in its domain. Finally, a Schur complement matrix is formed for the rows and columns corresponding...
to the separators and fed to a parallel direct solver. The Schur complement matrix is sparsified before the direct solution using a drop tolerance that is derived from $\tau$ (DPARM(14)). DPARM(25) allows the user to control drop tolerance for the sparsification of the Schur complement. The drop tolerance computed by WISMP is multiplied by DPARM(25). The default value of DPARM(25) is 1.0. An input value greater that 1.0 would result in a sparser Schur complement and a value less that 1.0 would result in a denser Schur complement.

- DPARM(26) or dparm[25], 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) or dparm[16], type O:
  When the conjugate gradient method is used, 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.

5 Miscellaneous Routines

In this section, we describe some optional routines available to the users for managing memory allocation, data distribution, and some other miscellaneous tasks. Just like other WSMP routines, these can be called from a C program by passing the arguments by reference (Note 4.2).

Note 5.1 Some routines in this section have underscores in their names, and due to different mangling conventions followed by different compilers, you may get an “undefined symbol” error while using one of these routines. Placing an explicit underscore at the end of the routine name usually fixes the problem. For example, if WS_SORTINDICES_I does not work, then try using WS_SORTINDICES_I_

5.1 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 of 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 4.4. The description of INFO is similar to that of IPARM(64).

Please read Note 5.1 at the beginning of this section.

5.2 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 4.4. The description of INFO is similar to that of IPARM(64).

Please read Note 5.1 at the beginning of this section.

5.3 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 5.1 at the beginning of this section.

5.4 WSETMAXTHRDS ( NUMTHRDS )

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

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

Note that if this routine is used, it must be called before the first call to any WSMP or PWSMP computational routine or the initialization routines (Section 5.10). Once WSMP/PWSMP is initialized, the number of threads cannot be changed for a given run.

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

5.5 WSSYSTEMSCOPE and WSPROCESSSCOPE

A call to WSSYSTEMSCOPE can be used to set the contention scope of threads to PTHREAD_SCOPE_SYSTEM. Similarly, WSPROCESSSCOPE can be called to set the contention scope of threads to PTHREAD_SCOPE_PROCESS. If these routines are used, they must be called before the first call to any WSMP or PWSMP computational routine or the initialization routines (Section 5.10). Currently, the default contention scope of the threads is PTHREAD_SCOPE_SYSTEM.

5.6 WSETMAXSTACK (FSTK)

All threads spawned by WSMP are, by default, assigned a 1 Mbyte stack in 32-bit mode and 4 Mbytes in 64-bit mode. In rare case, for very large matrices, this may not be enough for one or more threads. The user can increase or decrease the default stack size by calling WSETMAXSTACK prior to any computational or initialization routine of WSMP. The double precision input parameter FSTK determines the factor by which the default stack size of each thread is changed; e.g., if FSTK is 2.d0, then each thread is spawned with a 2 Mbyte stack in 32-bit mode and 8 Mbyte stack in 64-bit mode. If this routine is used, it must be called before the first call to any WSMP or PWSMP computational routine or the initialization routines (Section 5.10). In the distributed-memory parallel version, this routine, if used, must be called by all processes (it is effective on only those processes on which it is called).

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

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

5.7 WSETLF (DLF)\textsuperscript{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 5.10).

5.8 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 WSETNOBIGNMAL before initializing or calling any computational routine of WSMP or PWSMP.

5.9 \textbf{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 5.1 at the beginning of this section.

5.10 \textbf{WSMP\_INITIALIZE ( )\textsuperscript{S,T} and PWSMP\_INITIALIZE ( )\textsuperscript{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 5.4), WSSYSTEMSCOPE, WSPROCESSSCOPE (Section 5.5), WSETMAXSTACK (Section 5.6), WSETLF (Section 5.7), and WSETNOBIGNMAL (Section 5.8) 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 PWSMP.

Please read Note 5.1 at the beginning of this section.

5.11 \textbf{WSMP\_CLEAR ( )\textsuperscript{S,T} and PWSMP\_CLEAR ( )\textsuperscript{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 5.1 at the beginning of this section.

5.12 \textbf{WISFREE ( )\textsuperscript{S,T} and PWISFREE ( )\textsuperscript{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).

6 \textbf{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 \textit{double complex} or \textit{complex*16}. The WSMP web page at \url{http://www.research.ibm.com/projects/wsmp} contains example programs illustrating the use of this routine.

7 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 \textit{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.

8 Acknowledgements

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

References


