MUltifrontal Massively Parallel Solver (MUMPS Version 4.3.1) Users' guide *

P. R. Amestoy[†], I. S. Duff[‡], J.-Y. L'Excellent[§] and J. Koster[¶]

November 12, 2003

Abstract

This document describes the Fortran 90 and C user interface to MUMPS Version 4.3. We describe in detail the data structures, parameters, calling sequences, and error diagnostics. Example programs using MUMPS are also given.

*Information on how to obtain updated copies of MUMPS can be obtained from the Web page http://www.enseeiht.fr/apo/MUMPS/ or by sending email to mumps@cerfacs.fr

[†] amestoy@enseeiht.fr, ENSEEIHT-IRIT, 2 rue Camichel, Toulouse, France.

[‡] I.Duff@rl.ac.uk, Rutherford Appleton Laboratory, Chilton, Didcot, Oxon, OX11 0QX England, and CERFACS, Toulouse, France.

[§] Jean-Yves.L.Excellent@ens-lyon.fr, INRIA Rhône-Alpes, LIP-ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France.

[¶] jak@ii.uib.no, Parallab, University of Bergen, Norway.

Contents

1 Introduction

MUMPS ("MUltifrontal Massively Parallel Solver") is a package for solving linear systems of equations $\mathbf{Ax} = \mathbf{b}$, where the matrix $\mathbf{A}$ is sparse and can be either unsymmetric, symmetric positive definite, or general symmetric. MUMPS uses a multifrontal technique which is a direct method based on either the LU or the LDL^T factorization of the matrix. We refer the reader to the papers [3, 4, 7, 15, 16] for full details of the techniques used. MUMPS exploits both parallelism arising from sparsity in the matrix $\mathbf{A}$ and from dense factorizations kernels.

The main features of the MUMPS package include the solution of the transposed system, input of the matrix in assembled format (distributed or centralized) or elemental format, error analysis, iterative refinement, scaling of the original matrix, and return of a Schur complement matrix. MUMPS offers several built-in ordering algorithms, a tight interface to some external ordering packages such as PORD [18] and METIS [17], and the possibility for the user to input a given ordering. Finally, MUMPS is available in various arithmetics (real or complex, single or double).

The software is written in Fortran 90 although a C interface is available (see Section 8). The parallel version of MUMPS requires MPI for message passing and makes use of the BLAS [11, 12], BLACS, and ScaLAPACK [9] libraries. The sequential version only relies on BLAS.

MUMPS has been tested on an SGI Origin 2000, a CRAY T3E, an IBM SP, and a cluster of PC under Linux, and on the following operating systems: IRIX 6.4 and higher, UNICOS, AIX 4.3 and higher, and Linux.

MUMPS distributes the work tasks among the processors, but an identified processor (the host) is required to perform most of the analysis phase, distribute the incoming matrix to the other processors (slaves) in the case where the matrix is centralized, and collect the solution. The system $\mathbf{Ax} = \mathbf{b}$ is solved in three main steps:

1. Analysis. The host performs an ordering (see Section 2.2) based on the symmetrized pattern $\mathbf{A} + \mathbf{A}^{\mathrm{T}}$, and carries out symbolic factorization. A mapping of the multifrontal computational graph is then computed, and symbolic information is transferred from the host to the other processors. Using this information, the processors estimate the memory necessary for factorization and solution.
2. Factorization. The original matrix is first distributed to processors that will participate in the numerical factorization. The numerical factorization on each frontal matrix is conducted by a *master* processor (determined by the analysis phase) and one or more *slave* processors (determined dynamically). Each processor allocates an array for contribution blocks and factors; the factors must be kept for the solution phase.
3. Solution. The right-hand side $\mathbf{b}$ is broadcast from the host to the other processors. These processors compute the solution $\mathbf{x}$ using the (distributed) factors computed during Step 2, and the solution is assembled on the host.

Each of these phases can be called independently and several instances of MUMPS can be handled simultaneously. MUMPS allows the host processor to participate in computations during the factorization and solve phases, just like any other processor (see Sec. 2.8).

For both the symmetric and the unsymmetric algorithms used in the code, we have chosen a fully asynchronous approach with dynamic scheduling of the computational tasks. Asynchronous communication is used to enable overlapping between communication and computation. Dynamic scheduling was initially chosen to accommodate numerical pivoting in the factorization. The other important reason for this choice was that, with dynamic scheduling, the algorithm can adapt itself at execution time to remap work and data to more appropriate processors. In fact, we combine the main features of static and dynamic approaches; we use the estimation obtained during the analysis to map some of the main computational tasks; the other tasks are dynamically scheduled at execution time. The main data structures (the original matrix and the factors) are similarly partially mapped according to the analysis phase.

2 Main functionalities of MUMPS 4.3

We describe here the main functionalities of the solver MUMPS. The user should refer to Sections 4 and 5 for a complete description of the parameters that must be set or that are referred to in this Section.

The variables mentioned in this section are components of a structure `[sdcz]mumps_par` of type `[SDCZ]MUMPS_STRUC` (see Section 3) and for the sake of clarity, we refer to them only by their component name. For example, we use ICNTL to refer to `mumps_par%ICNTL`.

2.1 Input matrix structure

`MUMPS` provides several possibilities to input the matrix. This is controlled by the parameters ICNTL(5) and ICNTL(18).

The input matrix can be supplied in *elemental format* and must be input centrally on the host (ICNTL(5)=0 and ICNTL(18)=0). For implementation details see Section 4.5.

Otherwise, it can be supplied in *assembled format* (ICNTL(5)=0) in coordinate form, and, in this case, there are several possibilities (see Sections 4.4 and 4.6):

1. the matrix can be input centrally on the host processor (ICNTL(18)=0);
2. only the matrix structure is provided on the host for the analysis phase and the matrix entries are provided for the numerical factorization, distributed across the processors:
   - either according to a mapping supplied by the analysis (ICNTL(18)=1),
   - or according to a user determined mapping (ICNTL(18)=2);
3. it is also possible to distribute the matrix pattern and the entries in any distribution in local triplets (ICNTL(18)=3) for both analysis and factorization.

By default the input matrix is considered in assembled format (ICNTL(5)=0) and input centrally on the host processor (ICNTL(18)=0).

2.2 Symmetric orderings

A range of orderings to preserve sparsity is available in the analysis phase. Most of them have been introduced in release 4.2 of the `MUMPS` package. The parameter ICNTL(7) is used to control the ordering request.

Besides the approximate minimum degree ordering (AMD, [2]), an approximate minimum degree ordering with automatic quasi dense row detection (QAMD, [1]), an approximate minimum fill-in ordering (AMF), an ordering where bottom-up strategies are used to build separators by Jürgen Schulze at Paderborn (PORD, [18]), and the METIS package from Minnesota [17] are possible choices. For what concerns the METIS package, only their METIS_NODEND hybrid ordering routine can be used.

A user-supplied ordering can also be provided and the pivot order must be set by the user in PERM_IN (see Section 4.8). Also, it should be noted that the logic that handles this case is different from the built-in orderings so that, for example, a different performance and different internal data structures are created by a run that generates an ordering and a separate one that feeds that same ordering array in as input.

If ICNTL(7)=7, the `MUMPS` package will automatically choose the ordering depending on the ordering packages installed, the type of the matrix (symmetric or unsymmetric), the size of the matrix and the number of processors available.

The default value of ICNTL(7) is 7.

2.3 Other pre-processing facilities

Besides the symmetric orderings, `MUMPS` offers other pre-processing facilities: permuting to zero-free diagonal and prescaling.

Permutations to zero-free diagonal can be applied to very unsymmetric matrices and can help reduce fill-in and arithmetic. We use the public domain code MC64 [13, 14] to compute a column permutation. This functionality is controlled by ICNTL(6) and is inhibited for symmetric matrices.

Prescaling of the input matrix can help reduce fill-in during factorization and can improve the numerical accuracy. A range of classical scalings are provided and can be automatically performed before numerical factorization. This functionality is controlled by ICNTL(8).

For some values of ICNTL(6) and when ICNTL(8)ne0 the arrays COLSCA/ROWSCA are accessed (see Section 4.7)

2.4 Post-processing facilities

It has been shown [8] that with only two to three steps of iterative refinement the solution can often be significantly improved. Iterative refinement can be optionally performed after the solution step using the parameter ICNTL(10).

MUMPS also enables the user to perform classical error analysis based on the residuals (see the description of ICNTL(11) in Section 5). We calculate an estimate of the sparse backward error using the theory and metrics developed in [8]. We use the notation $\bar{\mathbf{x}}$ for the computed solution and a modulus sign on a vector or a matrix to indicate the vector or matrix obtained by replacing all entries by their moduli. The scaled residual

$$\frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{b}| + |\mathbf{A}||\bar{\mathbf{x}}|)_i} \tag{1}$$

is computed for all equations except those for which the numerator is nonzero and the denominator is small. For all the exceptional equations,

$$\frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{A}| \ |\bar{\mathbf{x}}|)_i + \|\mathbf{A}_i\|_\infty \|\bar{\mathbf{x}}\|_\infty} \tag{2}$$

is used instead, where $\mathbf{A}_i$ is row i of $\mathbf{A}$. The largest scaled residual (1) is returned, on the host, in RINFOG(7) and the largest scaled residual (2) is returned in RINFOG(8). If all equations are in category (1), zero is returned in RINFOG(8). The computed solution $\bar{\mathbf{x}}$ is the exact solution of the equation

$$(\mathbf{A} + \delta\mathbf{A})\mathbf{x} = (\mathbf{b} + \delta\mathbf{b}),$$

where

$$\delta\mathbf{A}_{ij} \le \max(\mathrm{RINFOG}(7), \mathrm{RINFOG}(8))|\mathbf{A}|_{ij},$$

and $\delta\mathbf{b}_i \le \max(\mathrm{RINFOG}(7)|\mathbf{b}|_i, \mathrm{RINFOG}(8)\|\mathbf{A}_i\|_\infty \|\bar{\mathbf{x}}\|_\infty)$. Note that $\delta\mathbf{A}$ respects the sparsity of $\mathbf{A}$. An upper bound for the error in the solution is returned in RINFOG(9). Finally condition numbers $cond_1$ and $cond_2$ for the matrix are returned in RINFOG(10) and RINFOG(11), respectively, and

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \mathrm{RINFOG}(7) \times cond_1 + \mathrm{RINFOG}(8) \times cond_2.$$

2.5 Solving the transposed system

Given a sparse matrix $\mathbf{A}$, the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ or $\mathbf{A}^{\mathbf{T}}\mathbf{x} = \mathbf{b}$ can be solved during the solve stage. This is controlled by ICNTL(9).

2.6 Return a specified Schur complement

A Schur complement matrix can be returned to the user by setting ICNTL(19) to a value different from zero. The user must specify the list of indices of the Schur matrix. MUMPS then provides both a partial factorization of the complete matrix and returns the assembled Schur matrix in user memory. The Schur matrix is considered as a full matrix. The partial factorization that builds the Schur matrix can also be used to solve linear systems associated with the "interior" variables.

For example, consider the partitioned matrix

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \tag{3}$$

where the variables of $\mathbf{A}_{2,2}$ are those specified by the user. Then the Schur complement, as returned by MUMPS, is $\mathbf{A}_{2,2} - \mathbf{A}_{2,1}\mathbf{A}_{1,1}^{-1}\mathbf{A}_{1,2}$, and the solve is performed on $\mathbf{A}_{1,1}$ only. (Entries in the solution vector corresponding to indices in the Schur matrix are explicitly set to 0.)

See also the descriptions of the components SIZE_SCHUR, LISTVAR_SCHUR, and SCHUR in Section 4.9.

Note that the Schur complement could be considered as an element contribution to the interface block in a domain decomposition and so MUMPS could be used to solve this subproblem using the element entry.

When the Schur complement option is chosen the numerical pivoting is suppressed and CNTL(1) is set to 0 (see Section 5). Moreover, the option to find a column permutation to have a zero-free diagonal is inhibited, i.e. ICNTL(6) is treated has 0.

2.7 Arithmetic versions

Several versions of the package MUMPS are available: REAL, DOUBLE PRECISION, COMPLEX, and DOUBLE COMPLEX.

This document applies to all four precisions. In the following we use the conventions below:

1. the term **real** is used for REAL or DOUBLE PRECISION,
2. the term **complex** is used for COMPLEX or DOUBLE COMPLEX,
3. complex version means either COMPLEX, or DOUBLE COMPLEX version,
4. real version means either REAL or DOUBLE PRECISION version.

2.8 The working host processor

The analysis phase is performed on the host processor. MUMPS allows the host to participate to computations during the factorization and solve phases, just like any other processor, by setting the variable PAR to 1 (see Section 4.2). This allows for example MUMPS to run on a single processor and avoids the host processor to be idle during the factorization and solve phases (as is the case for PAR=0). We thus generally recommend to use a working host processor (PAR=1).

The only case where it may be worth using PAR=0 is with a large centralized matrix on a purely distributed architecture with relatively small local memory: PAR=1 will lead to a memory imbalance because of storage related to the initial matrix on the host.

2.9 Sequential version

It is possible to use MUMPS sequentially by limiting the number of processors to one, but the link phase still requires the MPI, BLACS, and ScaLAPACK libraries and the user program needed to make explicit calls to MPI_INIT and MPI_FINALIZE.

A purely sequential version of MUMPS is also available: for this, a special library is distributed which provides all external symbols needed by MUMPS for a sequential environment. MUMPS can thus be used in a simple sequential program, ignoring anything related to MPI. Details on how to build a purely sequential version of MUMPS are available in the file README available in the MUMPS distribution. Note that for the sequential version, the component PAR must be set to 1 (see Section 4.2) and that the calling program should not make use of MPI.

2.10 Shared memory version

On networks of SMP nodes (multiprocessor nodes with a shared memory), a parallel shared memory BLAS library (also called multithread BLAS) is often provided by the manufacturer. Using shared memory BLAS (between 2 and 4 threads per MPI process) can be significantly more efficient than running with only MPI processes. For example on a computer with 2 SMP nodes and 16 processors per node, we advise to run using 16 MPI processes with 2 threads per MPI process.

3 Calling sequence

In the following we use the notation [SDCZ]MUMPS for referring to DMUMPS, SMUMPS, ZMUMPS or CMUMPS for REAL, DOUBLE PRECISION, COMPLEX and DOUBLE COMPLEX versions, respectively. Similarly [SDCZ]MUMPS_STRUC refers to either SMUMPS_STRUC, DMUMPS_STRUC, CMUMPS_STRUC, or ZMUMPS_STRUC, and [sdcz]mumps_struc.h to smumps_struc.h, dmumps_struc.h, cmumps_struc.h or zmumps_struc.h.

In the Fortran 90 interface, there is a single user callable subroutine per precision, called [SDCZ]MUMPS, that has a single parameter mumps_par of Fortran 90 derived datatype

[SDCZ]MUMPS_STRUC defined in [sdcz]mumps_struc.h. The interface is the same for the sequential version, only the compilation process and libraries need be changed. In the case of the parallel version, MPI must be initialized by the user before the first call to [SDCZ]MUMPS is made. The calling sequence for the DOUBLE PRECISION version may look as follows:

```
INCLUDE 'mpif.h'
INCLUDE 'dmumps_struc.h'
...
INTEGER IERR
TYPE (DMUMPS_STRUC) :: mumps_par
...
CALL MPI_INIT(IERR)     ! Not needed in purely sequential version
...
CALL DMUMPS( mumps_par )
...
CALL MPI_FINALIZE(IERR) ! Not needed in purely sequential version
```

For other precisions, dmumps_struc.h should be replaced by smumps_struc.h, cmumps_struc.h, or zmumps_struc.h, and the 'D' in DMUMPS and DMUMPS_STRUC by 'S', 'C' or 'Z'.

The variable mumps_par of datatype [SDCZ]MUMPS_STRUC holds all the data for the problem. It has many components, only some of which are of interest to the user. The other components are internal to the package. Some of the components must only be defined on the host. Others must be defined on all processors. The file [sdcz]mumps_struc.h defines the derived datatype and must always be included in the program that calls MUMPS. The file [sdcz]mumps_root.h, which is included in [sdcz]mumps_struc.h, must also be available at compilation time. Components of the structure [SDCZ]MUMPS_STRUC that are of interest to the user are shown in Figure 1.

The interface to MUMPS consists in calling the subroutine [SDCZ]MUMPS with the appropriate parameters set in mumps_par.

```
      INCLUDE '[sdcz]mumps_root.h'
      TYPE [SDCZ]MUMPS_STRUC
        SEQUENCE
C INPUT PARAMETERS
C *****************
C    Problem definition
C    ------------------
C    Solver (SYM=0 Unsymmetric, SYM=1 Sym. Positive Definite, SYM=2 General Symmetric)
C    Type of parallelism (PAR=1 host working, PAR=0 host not working)
         INTEGER SYM, PAR, JOB
C    Control parameters
C    ------------------
         INTEGER ICNTL(40)
         real CNTL(5)
         INTEGER N  ! Order of input matrix
C    Assembled input matrix : User interface
C    ----------------------------------------
         INTEGER NZ
         real/complex , DIMENSION(:), POINTER ::  A
         INTEGER, DIMENSION(:), POINTER :: IRN, JCN
C        Case of distributed matrix entry
C        --------------------------------
         INTEGER NZ_loc
         INTEGER, DIMENSION(:), POINTER :: IRN_loc, JCN_loc
         real/complex , DIMENSION(:), POINTER ::  A_LOC
C    Unassembled input matrix: User interface
C    ----------------------------------------
         INTEGER NELT
         INTEGER, DIMENSION(:), POINTER :: ELTPTR, ELTVAR
         real/complex , DIMENSION(:), POINTER ::  A_ELT
C    MPI Communicator
C    -----------------
         INTEGER COMM
C    Ordering and scaling, if given by user (optional)
C    -------------------------------------------------
         INTEGER, DIMENSION(:), POINTER :: PERM_IN
         real/complex, DIMENSION(:), POINTER ::  COLSCA, ROWSCA
C INPUT/OUTPUT data
C ******************
C    RHS : on input it holds the right-hand side
C          on output it always holds the assembled solution
C    -----------------------------------------------------
         real/complex, DIMENSION(:), POINTER ::  RHS
C OUTPUT data and Statistics
C **************************
         INTEGER, DIMENSION(:), POINTER :: SYM_PERM, UNS_PERM
         INTEGER INFO(40)
         real RINFO(20)
         real RINFOG(20) !  Global information (host only)
C    Schur
C    ------
         INTEGER SIZE_SCHUR
         INTEGER, DIMENSION(:), POINTER :: LISTVAR_SCHUR
         real/complex, DIMENSION(:), POINTER ::  SCHUR
C    Mapping potentially provided by MUMPS
C    ------------------------------------
         INTEGER, DIMENSION(:), POINTER :: MAPPING
      END TYPE [SDCZ]MUMPS_STRUC
```

Figure 1: Main components of the structure `[SDCZ]MUMPS_STRUC` defined in `[sdcz]mumps_struc.h`. **real/complex** qualifies parameters that are real in the real version and complex in the complex version, whereas **real** is used for parameters that are always real, even in the complex version of MUMPS.

4 Input and output parameters

In this section, we describe the components of the variable mumps_par% of datatype `[SDCZ]MUMPS_STRUC` that must be set by the user.

4.1 Control of the three main phases: Analysis, Factorization, Solve

mumps_par%**JOB** (integer) must be initialized by the user on all processors before a call to `MUMPS`. It controls the main action taken by `MUMPS`. It is not altered.

JOB=–1 initializes an instance of the package. This must be called before any other call to the package concerning that instance. It sets default values for other components of `MUMPS_STRUC`, which may then be altered before subsequent calls to `MUMPS`. Note that three components of the structure must always be set by the user (on all processors) before a call with JOB=–1. These are

- mumps_par%COMM,
- mumps_par%SYM, and
- mumps_par%PAR.

JOB=–2 destroys an instance of the package. All data structures associated with the instance, except those provided by the user in mumps_par, are deallocated. It should be called by the user only when no further calls to `MUMPS` with this instance are required. It should be called before a further JOB=–1 call with the same argument mumps_par.

JOB=1 performs the analysis. In this phase, `MUMPS` chooses pivots from the diagonal using a selection criterion to preserve sparsity. It uses the pattern of $\mathbf{A} + \mathbf{A}^T$ but ignores numerical values. It subsequently constructs subsidiary information for the numerical factorization (a JOB=2 call).

An option exists for the user to input the pivotal sequence (ICNTL(7)=1, see below) in which case only the necessary information for a JOB=2 call will be generated.

The numerical values of the original matrix, mumps_par%A, must be provided by the user during the analysis phase only for particular values of ICNTL(6) (computation of a column permutation to get a zero-free diagonal). See Section 5 for more details.

`MUMPS` uses the pattern of the matrix $\mathbf{A}$ input by the user. In the case of *a centralized matrix*, the following components of the structure defining the matrix pattern must be set by the user only on the host:

- mumps_par%N, mumps_par%NZ, mumps_par%IRN, and mumps_par%JCN if the user wishes to input the structure of the matrix in *assembled format* (ICNTL(5)=0 and ICNTL(18) $\neq$ 3) (see Section 4.4,
- mumps_par%N, mumps_par%NELT, mumps_par%ELTPTR, and mumps_par%ELTVAR if the user wishes to input the matrix in *elemental format* (ICNTL(5)=1) (see Section 4.5).

These components should be passed unchanged when later calling the factorization (JOB=2) and solve (JOB=3) phases.

In the case of *a distributed assembled matrix* (see Section 4.6 for more details and options),

- If ICNTL(18) = 1 or 2, the previous requirements hold except that IRN and JCN are no longer required and need not be passed unchanged to the factorization phase.
- If ICNTL(18) = 3, the user should provide
  - mumps_par%N on the host
  - mumps_par%NZ_loc, mumps_par%IRN_loc and mumps_par%JCN_loc on all slave processors. Those should be passed unchanged to the factorization (JOB=2) and solve (JOB=3) phases.

A call to `MUMPS` with JOB=1 must be preceded by a call with JOB=–1 on the same instance.

JOB=2 performs the factorization. It uses the numerical values of the matrix $\mathbf{A}$ provided by the user and the information from the analysis phase (JOB=1) to factorize the matrix $\mathbf{A}$.

If the matrix is centralized on the host (ICNTL(18)=0), the pattern of the matrix should be passed unchanged since the last call to the analysis phase (see JOB=1); the following components of the structure define the numerical values and must be set by the user (on the host only) before a call with JOB=2:

- mumps_par%A if the matrix is in assembled format (ICNTL(5)=0), or
- mumps_par%A_ELT if the matrix is in elemental format (ICNTL(5)=1).

If the initial matrix is distributed (ICNTL(5)=0 and ICNTL(18) $\neq$ 0), then the following components of the structure must be set by the user on all slave processors before a call with JOB=2:

- mumps_par%A_loc on all slave processors, and
- mumps_par%NZ_loc, mumps_par%IRN_loc and mumps_par%JCN_loc if ICNTL(18)=1 or 2. (For ICNTL(18)=3, NZ_loc, IRN_loc and JCN_loc have already been passed to the analysis step and must be passed unchanged.)

(See Sections 4.4–4.5–4.6.) The actual pivot sequence used during the factorization may differ slightly from the sequence returned by the analysis if the matrix $\mathbf{A}$ is not diagonally dominant. An option exists for the user to input scaling vectors or let MUMPS compute such vectors automatically (in arrays COLSCA/ROWSCA, ICNTL(8) $\neq$ 0, see Section 4.7).
A call to MUMPS with JOB=2 must be preceded by a call with JOB=1 on the same instance.

JOB=3 performs the solution. It uses the right-hand side **b** provided by the user and the factors generated by the factorization (JOB=2) to solve a system of equations $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$. The pattern and values of the matrix should be passed unchanged since the last call to the factorization phase (see JOB=2). The structure component mumps_par%RHS must be set by the user (on the host only) before a call with JOB=3. (See Section 4.11.)
A call to MUMPS with JOB=3 must be preceded by a call with JOB=2 (or JOB=4) on the same instance.

JOB=4 combines the actions of JOB=1 with those of JOB=2. It must be preceded by a call to MUMPS with JOB=–1 on the same instance.

JOB=5 combines the actions of JOB=2 and JOB=3. It must be preceded by a call to MUMPS with JOB=1 on the same instance.

JOB=6 combines the actions of calls with JOB=1, 2, and 3. It must be preceded by a call to MUMPS with JOB=–1 on the same instance.

Consecutive calls with JOB=2,3,5 on the same instance are possible.

4.2 Control of parallelism

mumps_par%**COMM** (integer) must be set by the user on all processors before the initialization phase (JOB=–1) and must not be changed. It must be set to a valid MPI communicator that will be used for message passing inside MUMPS. It is not altered by MUMPS. The processor with rank 0 in this communicator is used by MUMPS as the **host** processor.

mumps_par%**PAR** (integer) must be initialized by the user on all processors and is accessed by MUMPS only during the initialization phase (JOB=–1). It is not altered by MUMPS. Possible values for PAR are:

0 host is not involved in factorization/solve phases

1 host is involved in factorization/solve phases

Other values are treated as 1.

If PAR is set to 0, the host will only hold the initial problem, perform symbolic computations during the analysis phase, distribute data, and collect results from other processors. If set to 1, the host will also participate in the factorization and solve phases. If the initial problem is large and memory is an issue, PAR = 1 is not recommended if the matrix is centralized on processor 0 because this can lead to memory imbalance, with processor 0 having a larger memory load than the other processors. Note that setting PAR to 1, and using only 1 processor, leads to a sequential code.

4.3 Matrix type

mumps_par%**SYM** (integer) must be initialized by the user on all processors and is accessed by MUMPS only during the initialization phase (JOB=–1). It is not altered by MUMPS except for the complex version of MUMPS where SYM=1 is replaced by SYM=2 and structural symmetry is exploited up to the root. Possible values for SYM are:

0 **A** is unsymmetric

1 **A** is symmetric positive definite

2 **A** is general symmetric

For the complex version, the value SYM=1 is currently treated as SYM=2. We do not have a version for Hermitian matrices in this release of MUMPS.

4.4 Centralized assembled matrix input: ICNTL(5)=0 and ICNTL(18)=0

mumps_par%N (integer), mumps_par%NZ (integer), mumps_par%IRN (integer array pointer, dimension NZ), mumps_par%JCN (integer array pointer, dimension NZ), and mumps_par%A (**real/complex** array pointer, dimension NZ) hold the matrix in assembled format. These components should be set by the user only on the host and only when ICNTL(5)=0 and ICNTL(18)=0:

- N is the order of the matrix **A**, N $>$ 0. It is not altered by MUMPS.
- NZ is the number of entries being input, NZ $>$ 0. It is not altered by MUMPS.
- IRN, JCN are integer arrays of length NZ containing the row and column indices, respectively, for the matrix entries. IRN is unchanged. JCN is unchanged unless ICNTL(6)$>$0, in which case the original matrix might be permuted to have a zero-free diagonal.
- A is a **real** (**complex** in the complex version) array of length NZ. The user must set A(k) to the value of the entry in row IRN(k) and column JCN(k) of the matrix. A is accessed when JOB=1 only when ICNTL(6)neq0. Duplicate entries are summed and any with IRN(k) or JCN(k) out-of-range are ignored.
  Note that, in the case of the symmetric solver, a diagonal nonzero a_{ii} is held as A(k)=a_{ii}, IRN(k)=JCN(k)=i, and a pair of off-diagonal nonzeros $a_{ij} = a_{ji}$ is held as A(k)=a_{ij} and IRN(k)=i, JCN(k)=j or vice-versa. Again, duplicate entries are summed and entries with IRN(k) or JCN(k) out-of-range are ignored.

The components N, NZ, IRN, and JCN describe the pattern of the matrix and must be set by the user before the analysis phase (JOB=1). Component A must be set before the factorization phase (JOB=2).

4.5 Element matrix input: ICNTL(5)=1 and ICNTL(18)=0

mumps_par%N (integer), mumps_par%NELT (integer), mumps_par%ELTPTR (integer array pointer, dimension NELT+1), mumps_par%ELTVAR (integer array pointer, dimension ELTPTR(NELT+1)–1), and mumps_par%A_ELT (**real/complex** array pointer) hold the matrix in elemental format. These components should be set by the user only on the host and only when ICNTL(5)=1:

- N is the order of the matrix **A**, N $>$ 0. It is not altered by MUMPS.
- NELT is the number of elements being input, NELT $>$ 0. It is not altered by MUMPS.
- ELTPTR is an integer array of length NELT+1. ELTPTR(j) points to the position in ELTVAR of the first variable in element j, and ELTPTR(NELT+1) must be set to the position after the last variable of the last element. Note that ELTPTR(1) should be equal to 1. It is not altered by MUMPS.
- ELTVAR is an integer array of length ELTPTR(NELT+1)–1 and must be set to the lists of variables of the elements. It is not altered by MUMPS. Those for element j are stored in positions ELTPTR(j), ..., ELTPTR(j+1)–1. Out-of-range variables are ignored.
- A_ELT is a **real** (**complex** in the complex version) array. If N_{p} denotes ELTPTR(p+1)–ELTPTR(p), then the values for element j are stored in positions $K_{\mathrm{j}}+1, \ldots, K_{\mathrm{j}}+L_{\mathrm{j}}$, where
  – $K_{\mathrm{j}} = \sum_{\mathrm{p}=1}^{\mathrm{j}-1} {N_{\mathrm{p}}}^2$, and $L_{\mathrm{j}} = {N_{\mathrm{j}}}^2$ in the unsymmetric case (SYM = 0)

– $K_j = \sum_{p=1}^{j-1}(N_p \cdot (N_p + 1))/2$, and $L_j = (N_j \cdot (N_j + 1))/2$ in the symmetric case (SYM $\neq$ 0). Only the lower triangular part is stored.

Values within each element are stored column-wise. Values corresponding to out-of-range variables are ignored and values corresponding to duplicate variables within an element are summed. A_ELT is not accessed when JOB = 1. Note that, although the elemental matrix may be symmetric or unsymmetric in value, its structure is always symmetric.

The components N, NELT, ELTPTR, and ELTVAR describe the pattern of the matrix and must be set by the user before the analysis phase (JOB=1). Component A_ELT must be set before the factorization phase (JOB=2). Note that, in the current release of the package, the element entry must be centralized on the host.

4.6 Distributed assembled matrix input: ICNTL(5)=0 and ICNTL(18)≠0

When the matrix is in assembled form (ICNTL(5)=0), we offer several options, defined by the control parameter ICNTL(18) described in Section 5. The following components of the structure define the distributed assembled matrix input. They are valid for nonzero values of ICNTL(18), otherwise the user should refer to Section 4.4.

mumps_par%N (integer), mumps_par%NZ (integer), mumps_par%IRN (integer array pointer, dimension NZ), mumps_par%JCN (integer array pointer, dimension NZ), mumps_par%IRN_loc (integer array pointer, dimension NZ_loc), mumps_par%JCN_loc (integer array pointer, dimension NZ_loc), mumps_par%A_loc (**real/complex** array pointer, dimension NZ_loc), and mumps_par%MAPPING (integer array, dimension NZ).

- N is the order of the matrix **A**, N > 0. It must be set on the host before analysis. It is not altered by MUMPS.
- NZ is the number of entries being input in the definition of **A**, NZ > 0. It must be defined on the host before analysis if ICNTL(18) = 1, or 2.
- IRN, JCN are integer arrays of length NZ containing the row and column indices, respectively, for the matrix entries. They must be defined on the host before analysis if ICNTL(18) = 1, or 2. They can be deallocated by the user just after the analysis.
- NZ_loc is the number of entries local to a processor. It must be defined on all processors in the case of the working host model of parallelism (PAR=1), and on all processors except the host in the case of the non-working host model of parallelism (PAR=0), before analysis if ICNTL(18) = 3, and before factorization if ICNTL(18) = 1 or 2.
- IRN_loc, JCN_loc are integer arrays of length NZ_loc containing the row and column indices, respectively, for the matrix entries. They must be defined on all processors if PAR=1, and on all processors except the host if PAR=0, before analysis if ICNTL(18) = 3, and before factorization if ICNTL(18) = 1 or 2.
- A_loc is a **real** (**complex** in the complex version) array of dimension NZ_loc that must be defined before the factorization phase (JOB=2) on all processors if PAR = 1, and on all processors except the host if PAR = 0. The user must set A_loc(k) to the value in row IRN_loc(k) and column JCN_loc(k).
- MAPPING is an integer array of size NZ which is returned by MUMPS on the host after the analysis phase as an indication of a preferred mapping if ICNTL(18) = 1. In that case, MAPPING(i) = IPROC means that entry IRN(i), JCN(i) should be provided on processor with rank IPROC in the MUMPS communicator.

We recommend the use of options ICNTL(18)= 2 or 3 because they are the simplest and most flexible options. Furthermore, those options (2 or 3) are in general almost as efficient as the more sophisticated (but more complicated for the user) option ICNTL(18)=1.

4.7 Prescaling

mumps_par%**COLSCA**, mumps_par%**ROWSCA** (double precision array pointers, dimension N) are optional scaling arrays required only by the host. If a scaling is provided by the user

(ICNTL(8)=–1), these arrays must be allocated and initialized by the user on the host, before a call to the factorization phase (JOB=2). They might also be automatically allocated and computed by the package during analysis (if ICNTL(6)=5 or 6). They should be passed unchanged to the solve phase (JOB=3).

4.8 Given ordering

mumps_par%**PERM_IN** (integer array pointer, dimension N) must be allocated and initialized by the user on the host if ICNTL(7)=1. It is accessed during the analysis (JOB=1) and PERM_IN(i), i=1, . . . , N must hold the position of variable i in the pivot order. Note that, even when the ordering is provided by the user, the analysis must still be performed before numerical factorization.

4.9 Return a Schur complement

mumps_par%**SIZE_SCHUR** (integer) must be initialized on the host to the size of the Schur complement if ICNTL(19) $\neq$ 0. It is accessed during the analysis phase and should be passed unchanged to the factorization and solve phases.

mumps_par%**LISTVAR_SCHUR** (integer array pointer, dimension mumps_par%**SIZE_SCHUR** must be allocated and initialized by the user on the host if ICNTL(19) $\neq$ 0. It is not altered by MUMPS. It is accessed during analysis (JOB=1) and LISTVAR_SCHUR(i), i=1, . . . , SIZE_SCHUR must hold the i^{th} index of the Schur matrix.

mumps_par%**SCHUR** is a **real** (**complex** in the complex version) pointer array of size SIZE_SCHUR $\times$ SIZE_SCHUR that must be allocated by the user on the host before the factorization phase if ICNTL(19) $\neq$ 0. On exit, it holds the Schur complement matrix (see ICNTL(19) above).

4.10 Workspace parameters

mumps_par%**MAXIS** and mumps_par%**MAXS** (integers) are defined, for each processor, as the size of the integer and the real (complex for the complex version) workspaces respectively required for factorization and/or solve. On return from analysis (JOB = 1), INFO(7) and INFO(8) return the minimum values for MAXIS and MAXS, respectively, to the user. If the user has reason to believe that significant numerical pivoting will be required, it may be desirable to choose a higher value for MAXIS (or MAXS) than output from the analysis. At the beginning of the factorization, MAXIS and MAXS are set to the maximum of estimates based on analysis phase data and the values supplied by the user. An integer array IS of size MAXIS and a real (complex in the complex version) array S of size MAXS are then dynamically allocated and used during the factorization and solve phases to hold the factors and contribution blocks.

4.11 Right-hand side and solution vector

mumps_par%**RHS** (**real/complex** array pointer, dimension N) is a **real** (**complex** in the complex version) array that must be set by the user on the host only, before a call to MUMPS with JOB = 3, 5, or 6. On entry, RHS(i) must hold the i-th component of the right-hand side of the equations being solved. On exit, RHS(i) will hold the i-th component of the solution vector.

5 Control parameters

On exit from the initialization call (JOB=–1), the control parameters are set to default values. If the user wishes to use values other than the defaults, the corresponding entries in mumps_par%ICNTL and mumps_par%CNTL should be reset after this initial call and before the call in which they are used.

mumps_par%**ICNTL** is an integer array of dimension 40.

ICNTL(1) is the output stream for error messages. If it is negative or zero, these messages will be suppressed. Default value is 6.

ICNTL(2) is the output stream for diagnostic printing, statistics, and warning messages. If it is negative or zero, these messages will be suppressed. Default value is 0.

ICNTL(3) is the output stream for global information, collected on the host. If it is negative or zero, these messages will be suppressed. Default value is 6.

ICNTL(4) is the level of printing for error, warning, and diagnostic messages. Maximum value is 4 and default value is 2 (errors and warnings printed). Possible values are

- ≤ 0: No messages output.
- 1 : Only error messages printed.
- 2 : Errors and warnings printed.
- 3 : Errors and warnings and terse diagnostics (only first ten entries of arrays) printed.
- 4 : Errors and warnings and all information on input and output parameters printed.

ICNTL(5) has default value 0 and is only accessed by the host and only during the analysis phase. If ICNTL(5) = 0, the input matrix must be given in assembled format in the structure components N, NZ, IRN, JCN, and A (or NZ_loc, IRN_loc, JCN_loc, A_loc, see Section 4.6). If ICNTL(5) = 1, the input matrix must be given in elemental format in the structure components N, NELT, ELTPTR, ELTVAR, and A_ELT.

ICNTL(6) has default value 7 for unsymmetric matrices and 0 for symmetric matrices. It is only accessed by the host and only during the analysis phase. If ICNTL(6)=1, 2, 3, 4, 5, 6, 7 a column permutation based on the public domain code MC64 (see [13, 14] for more details) is applied to the original matrix. Column permutations are then applied to the original matrix to get a zero-free diagonal. Possible values of ICNTL(6) are:

- 0 : No column permutation is computed.
- 1 : The permuted matrix has as many entries on its diagonal possible. The values on the diagonal are of arbitrary size.
- 2 : The smallest value on the diagonal of the permuted matrix is maximized.
- 3 : Variant of option 2 with different performance.
- 4 : The sum of the diagonal entries of the permuted matrix is maximized.
- 5 : The product of the diagonal entries of the permuted matrix is maximized. Vectors are also computed (and stored in COLSCA and ROWSCA, only if ICNTL(8) was set to 7) to scale the permuted matrix so that the nonzero diagonal entries in the permuted matrix are one in absolute value and all the off-diagonal entries are less than or equal to one in absolute value.
- 6 : Similar to 5 but with a different algorithm.
- 7 : Based on the structural symmetry of the input matrix and on the availability of the numerical values, the value of ICNTL(6) is automatically chosen by the software.

Other values are treated as 0.

Except for ICNTL(6)=0 or 1, the numerical values of the original matrix, mumps_par%A, must be provided by the user during the analysis phase. The user is advised to set ICNTL(6) only when the matrix is very unsymmetric. If the matrix is symmetric (SYM $\neq 0$), or in elemental format (ICNTL(5)=1), or the ordering is provided by the user (ICNTL(7)=1), or the Schur option (ICNTL(19) $\neq 0$) is required, or the matrix is initially distributed (ICNTL(18) $\neq 0$) then ICNTL(6) is treated as zero. On output from the analysis phase, when the column permutation is not the identity, the pointer mumps_par%UNS_PERM (internal data valid until a call to MUMPS with JOB=-2) provides access to the permutation. Otherwise, the pointer is unassociated.

ICNTL(7) has default value 7 and is only accessed by the host and only during the analysis phase. It determines the pivot order to be used for the factorization. Note that, even when the ordering is provided by the user, the analysis must be performed before numerical factorization. Possible values are:

- 0 : Approximate Minimum Degree (AMD) [2] is used,
- 1 : the pivot order should be set by the user in PERM_IN. In this case, PERM_IN(i), (i=1, ... N) holds the position of variable i in the pivot order.

- 2 : the Approximate Minimum Fill (AMF) is used,
- 3 : Not available in the current version.
- 4 : PORD[1] [18] is used,
- 5 : the METIS[2] [17] routine METIS_NODEND is used,
- 6 : the Approximate Minimum Degree with automatic quasi-dense row detection (QAMD) is used.
- 7 : Automatic value chosen by the software during analysis phase. This choice will depend on the ordering packages made available, on the matrix (type and size), and on the number of processors.

Other values are treated as 7. Currently, options 3, 4 and 5 are only available if the corresponding packages are installed (see comments in the Makefiles to let MUMPS know about them). If the packages are not installed or if the matrix is by elements, options 3, 4 and 5 are treated as 7.

With option 7, the automatic value of ICNTL(7) chosen by the package depends on the ordering packages installed, the type of matrix (symmetric or unsymmetric), the size of the matrix and the number of processors.

For linear programming matrices of form $\mathbf{AA}^T$, and for matrices with relatively dense rows, we highly recommend option 6 which may significantly reduce the time for analysis.

If the user asks for a Schur complement matrix, or if the matrix is by elements, only options 0 and 1 are currently available. On output, the pointer mumps_par%SYM_PERM (internal data valid until a call to MUMPS with JOB=-2) provides access to the symmetric permutation.

ICNTL(8) has default value 0 for symmetric matrices and 7 for unsymmetric matrices. It is used to describe the scaling strategy and is only accessed by the host.

On entry to the analysis phase, if ICNTL(8) = 7, then an automatic choice of the scaling option is performed during the analysis and ICNTL(8) is modified accordingly. In particular, if ICNTL(8) is reset to -1 by the package during the analysis, scaling arrays have been computed internally and are ready to be used by the factorization phase. This corresponds to the case where ICNTL(6) was equal 5, 6, or 7.

On entry to the factorization phase, if ICNTL(8) = –1, scaling vectors must be provided in COLSCA and ROWSCA (either by the package, see previous paragraph, either by the user, who is then responsible for allocating and freeing them). If ICNTL(8) = 0, no scaling is performed, and arrays COLSCA/ROWSCA are not used. If ICNTL(8) $>$ 0, the scaling arrays COLSCA/ROWSCA are allocated and computed by the package during the factorization phase.

Possible values of ICNTL(8) are listed below:

- -1: Scaling provided on entry to numerical fatorization phase,
- 0 : No scaling applied/computed.
- 1 : Diagonal scaling,
- 2 : Scaling based on [10] (HSL code MC29),
- 3 : Column scaling,
- 4 : Row and column scaling,
- 5 : Scaling based on [10] followed by column scaling,
- 6 : Scaling based on [10] followed by row and column scaling.
- 7 (analysis only) : Automatic choice of scaling value done during analysis.

If the input matrix is symmetric (SYM $\neq$ 0), then only options –1, 0, and 1 are allowed and other options are treated as 0; if ICNTL(8)=–1, the user should ensure that the array ROWSCA is equal to the array COLSCA. If the input matrix is in elemental format (ICNTL(5) = 1), then only options –1 and 0 are allowed and other options are treated as 0. If the initial matrix is distributed (ICNTL(18) $\neq$ 0 and ICNTL(5) = 0) or if rank-revealing options are set (ICNTL(16) $\neq$ 0), then the value of ICNTL(8) is ignored and no scaling is applied.

[1]Distributed within MUMPS by permission of J. Schulze (University of Paderborn).

[2]See http://www-users.cs.umn.edu/∼karypis/metis/ to obtain a copy.

ICNTL(9) has default value 1 and is only accessed by the host during the solve phase. If ICNTL(9) = 1, $\mathbf{Ax} = \mathbf{b}$ is solved, otherwise, $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ is solved.

ICNTL(10) has default value 0 and is only accessed by the host during the solve phase. It corresponds to the maximum number of steps of iterative refinement. If ICNTL(10) $\leq$ 0, iterative refinement is not performed.

ICNTL(11) has default value 0 and is only accessed by the host and only during the solve phase. A positive value will return statistics related to the linear system solved ($\mathbf{Ax} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ depending on the value of ICNTL(9)): the infinite norm of the input matrix, the computed solution, and the scaled residual in RINFOG(4) to RINFOG(6), respectively, a backward error estimate in RINFOG(7) and RINFOG(8), an estimate for the error in the solution in RINFOG(9), and condition numbers for the matrix in RINFOG(10) and RINFOG(11). See also Section 2.4. Note that if performance is concerned, ICNTL(11) should be left to 0.

Note that, although the following ICNTL entries (12 to 14) control the efficiency of the factorization and solve phases, they involve preprocessing work performed during analysis and must thus be set at the analysis phase.

ICNTL(12) has default value 0 and is only accessed by the host and only during the analysis phase. If ICNTL(12) = 0, node level parallelism is switched on, otherwise only tree parallelism will be used during factorization/solve phases.

ICNTL(13) has default value 0 and is only accessed by the host during the analysis phase. If ICNTL(13) = 0, ScaLAPACK will be used for the root node if the size of the root node of the assembly tree is larger than a machine-dependent minimum size. Otherwise, the root node of the tree will be processed sequentially.

ICNTL(14) is accessed by the host both during the analysis and the factorization phases. It corresponds to the percentage increase in the estimated working space. When significant extra fill-in is caused by numerical pivoting, larger values of ICNTL(14) may help use the real working space more efficiently. Default value is 20 % except for symmetric positive definite matrices (SYM=1) where the default value is 15 %.

ICNTL(15-17) Experimental rank-revealing functionalities, available on request.

ICNTL(18) has default value 0 and is only accessed by the host during the analysis phase, if the matrix format is assembled (ICNTL(5) = 0). ICNTL(18) defines the strategy for the distributed input matrix. Possible values are:

- 0: the input matrix is centralized on the host. This is the default, see Section 4.4.
- 1: the user provides the structure of the matrix on the host at analysis, `MUMPS` returns a mapping and the user should then provide the matrix distributed according to the mapping on entry to the numerical factorization phase.
- 2: the user provides the structure of the matrix on the host at analysis, and the distributed matrix on all slave processors at factorization. Any distribution is allowed.
- 3: user directly provides the distributed matrix input both for analysis and factorization.

For options 1, 2, 3, see Section 4.6 for more details on the input/output parameters to `MUMPS`. For flexibility, options 2 or 3 are recommended.

ICNTL(19) has default value 0 and is only accessed by the host during the analysis phase. If ICNTL(19) $\neq$ 0 then the Schur matrix will be returned to the user. The user must set on entry on the host node (before analysis):

- the integer variable SIZE_SCHUR to the size of the Schur matrix,
- the integer array pointer LISTVAR_SCHUR to the list of indices of the Schur matrix.

Before the factorization phase, on the host node, the 1-dimensional pointer array SCHUR should point to SIZE_SCHUR*SIZE_SCHUR locations in memory, allocated by the user. On output from the factorization phase, and on the host node, the 1-dimensional pointer array SCHUR of length SIZE_SCHUR*SIZE_SCHUR holds the (dense) Schur matrix of order SIZE_SCHUR. Note that the order of the indices in the Schur matrix is identical to the order provided by the user in

LISTVAR_SCHUR and that the Schur matrix is stored **by rows**. If the matrix is symmetric then only the lower triangular part of the Schur matrix is provided (**by rows**) and the upper part is not significant.

The partial factorization of the interior variables can then be exploited to perform a solve phase (transposed matrix or not). Note that the right-hand side (RHS) provided on input must still be of size N even if only the N-SIZE_SCHUR indices will be considered and if only N-SIZE_SCHUR indices of the solution will be relevant to the user.

Finally note that since the Schur complement can be viewed as a partial factorization of the global matrix (with partial ordering of the variables provided by the user) the following options of MUMPS are incompatible with the Schur option: maximum transversal, scaling, iterative refinement, error analysis. Note that if the ordering is given then the following property should hold: PERM_IN(LISTVAR_SCHUR(i)) = N-SIZE_SCHUR+i, for i=1,SIZE_SCHUR.

ICNTL(20-40) are not used in the current version.

mumps_par%**CNTL** is a **real** (also **real** in the complex version) array of dimension 5.

CNTL(1) is the relative threshold for numerical pivoting. It forms a trade-off between preserving sparsity and ensuring numerical stability during the factorization. In general, a larger value of CNTL(1) increases fill-in but leads to a more accurate factorization. If CNTL(1) is nonzero, numerical pivoting will be performed. If CNTL(1) is zero, no such pivoting will be performed and the subroutine will fail if a zero pivot is encountered. If the matrix is diagonally dominant, then setting CNTL(1) to zero will decrease the factorization time while still providing a stable decomposition. If the code is called for unsymmetric or general symmetric matrices, CNTL(1) has default value 0.01. For symmetric positive definite matrices and if the Schur complement is asked to be returned (ICNTL(19)$\neq$ 0), numerical pivoting is suppressed and the default value is 0.0. Values less than 0.0 are treated as 0.0, values greater than 1.0 are treated as 1.0.

CNTL(2) is the stopping criterion for iterative refinement and is only accessed by the host during the solve phase. Let $Berr = \max_i \frac{|r|_i}{(|A| \cdot |x| + |b|)_i}$ [8]. Iterative refinement will stop when either the required accuracy is reached ($Berr <$ CNTL(2)) or the convergence rate is too slow ($Berr$ does not decrease by at least a factor of 5). Default value is $\sqrt{\varepsilon}$.

CNTL(3) determines the absolute threshold $thres$ for numerical pivoting. It has default value -1.0 and is only accessed by the host during the numerical factorization phase. If CNTL(3) < 0 (default), $thres$ is determined automatically: $thres = \epsilon \|A\|$ if ICNTL(16) $\neq 0$ or if SYM=2 in the case of node level parallelism; $thres = 0$ otherwise. If CNTL(3) ≥ 0, then the value $thres =$ CNTL(3) is used. During the numerical factorization, a potential pivot has to be larger than $thres$ to be accepted.

CNTL(4) - CNTL(5) are not used in the current version.

6 Information parameters

The parameters described in this section are returned by MUMPS and hold information that may be of interest to the user. Some of the information is local to each processor and some only on the host. If an error is detected (see Section 7), the information may be incomplete.

6.1 Information local to each processor

The arrays mumps_par%**RINFO** and mumps_par%**INFO** are local to each process.

mumps_par%**RINFO** is a double precision array of dimension 20. It contains the following local information on the execution of MUMPS:

RINFO(1) - after analysis: The estimated number of floating-point operations on the processor for the elimination process.

RINFO(2) - after factorization: The number of floating-point operations on the processor for the assembly process.

RINFO(3) - after factorization: The number of floating-point operations on the processor for the elimination process.

RINFO(4) - RINFO(20) are not used in the current version.

mumps_par%**INFO** is an integer array of dimension 40. It contains the following local information on the execution of MUMPS:

INFO(1) is 0 if the call to MUMPS was successful, negative if an error occurred (see Section 7), or positive if a warning is returned.

INFO(2) holds additional information about the error or the warning. If INFO(1)=–1, INFO(2) is the processor number (in communicator mumps_par%COMM) on which the error was detected.

INFO(3) - after analysis: Estimated real space needed on the processor for factors.

INFO(4) - after analysis: Estimated integer space needed on the processor for factors.

INFO(5) - after analysis: Estimated maximum front size on the processor.

INFO(6) - after analysis: Number of nodes in the complete tree. The same value is returned on all processors.

INFO(7) - after analysis: Minimum value of MAXIS estimated by the analysis phase to run the numerical factorization successfully.

INFO(8) - after analysis: Minimum value of MAXS estimated by the analysis phase to run the numerical factorization successfully.

INFO(9) - after factorization: Size of the real space used on the processor to store the LU factors.

INFO(10) - after factorization: Size of the integer space used on the processor to store the LU factors.

INFO(11) - after factorization: Order of the largest frontal matrix processed on the processor.

INFO(12) - after factorization: Number of off-diagonal pivots encountered on the processor if SYM=0 or number of negative pivots on the processor if SYM=1 or 2. If ICNTL(13)=0 (the default), this excludes pivots from the parallel root node treated by ScaLAPACK. Note that if SYM=1 or 2, INFO(12) will be 0 for complex symmetric matrices.

INFO(13) - after factorization: The number of uneliminated variables, corresponding to delayed pivots, sent to the father. If a delayed pivot is subsequently passed to the father of the father, it is counted a second time.

INFO(14) - after factorization: Number of memory compresses on the processor.

INFO(15) - after analysis: estimated total size (in millions of bytes) of all MUMPS internal data for running numerical factorization.

INFO(16) - after factorization: total size (in millions of bytes) of all MUMPS internal data used during numerical factorization.

INFO(17) - INFO(40) are not used in the current version.

6.2 Information available on the host

The arrays mumps_par%RINFOG and mumps_par%INFOG :

mumps_par%**RINFOG** is a double precision array of dimension 20. It contains the following global information on the execution of MUMPS:

RINFOG(1) - after analysis: The estimated number of floating-point operations (on all processors) for the elimination process.

RINFOG(2) - after factorization: The total number of floating-point operations (on all processors) for the assembly process.

RINFOG(3) - after factorization: The total number of floating-point operations (on all processors) for the elimination process.

RINFOG(4) to RINFOG(11) - after solve with error analysis: Only returned on the host process if ICNTL(11) $\neq 0$. See description of ICNTL(11).

RINFOG(12) - RINFOG(20) are not used in the current version.

mumps_par%**INFOG** is an integer array of dimension 40. It contains the following global information on the execution of MUMPS:

INFOG(1) is 0 if the call to MUMPS was successful, negative if an error occurred (see Section 7), or positive if a warning is returned.

INFOG(2) holds additional information about the error or the warning.

The difference between INFOG(1:2) and INFO(1:2) is that INFOG(1:2) is the same on all processors. It has the value of INFO(1:2) of the processor which returned with the most negative INFO(1) value. For example, if processor p returns with INFO(1)=-13, and INFO(2)=10000, then all other processors will return with INFOG(1)=-13 and INFOG(2)=10000, but still INFO(1)=-1 and INFO(2)=p.

INFOG(3) - after analysis: Total estimated real workspace for factors on all processors.

INFOG(4) - after analysis: Total estimated integer workspace for factors on all processors.

INFOG(5) - after analysis: Estimated maximum front size in the complete tree.

INFOG(6) - after analysis: Number of nodes in the complete tree.

INFOG(7) - after analysis: ordering option effectively used (see ICNTL(7)).

INFOG(8) - after analysis: structural symmetry in percent (100 : symmetric, 0 : fully unsymmetric) of the (permuted) matrix. (-1 indicates that the structural symmetry was not computed.)

INFOG(9) - after factorization: Total real space to store the LU factors.

INFOG(10) - after factorization: Total integer space to store the LU factors.

INFOG(11) - after factorization: Order of largest frontal matrix.

INFOG(12) - after factorization: Total number of off-diagonal pivots if SYM=0 or total number of negative pivots if SYM=1 or 2. If ICNTL(13)=0 (the default) this excludes pivots from the parallel root node treated by ScaLAPACK. Note that if SYM=1 or 2, INFOG(12) will be 0 for complex symmetric matrices.

INFOG(13) - after factorization: Total number of delayed pivots.

INFOG(14) - after factorization: Total number of memory compresses.

INFOG(15) - after solution: Number of steps of iterative refinement.

INFOG(16) - after analysis: Estimated size (in million of bytes) of all MUMPS internal data for running factorization: value on the most memory consuming processor.

INFOG(17) - after analysis: Estimated size (in millions of bytes) of all MUMPS internal data for running factorization: sum over all processors.

INFOG(18) - after factorization: Size in millions of bytes of all MUMPS internal data allocated during factorization: value on the most memory consuming processor.

INFOG(19) - after factorization: Size in millions of bytes of all MUMPS internal data allocated during factorization: sum over all processors.

INFOG(20) - after analysis: Estimated number of entries in the factors.

INFOG(21) - INFOG(40) are not used in the current version.

7 Error diagnostics

MUMPS uses the following mechanism to process errors that may occur during the parallel execution of the code. If, during a call to MUMPS, an error occurs on a processor, this processor informs all the other processors before they return from the call. In parts of the code where messages are sent asynchronously (for example factorization and solve phases), the processor on which the error occurs sends a message to the other processors with a specific error tag. On the other hand, if the error occurs in a subroutine that does not use asynchronous communication, the processor propagates the error to the other processors.

On successful completion, a call to MUMPS will exit with the parameter mumps_par%INFOG(1) set to zero. A negative value for mumps_par%INFOG(1) indicates that an error has been detected on one of the processors. For example, if processor s returns with INFO(1)=–8 and INFO(2)=1000, then processor s ran out of integer workspace during the factorization and the size of the workspace MAXIS should be increased by 1000 at least. The other processors are informed about this error and return with INFO(1) = –1 (i.e., an error occurred on another processor) and INFO(2)=s (i.e., the error occurred on processor s). Processors that detected a local error, do not overwrite INFO(1), i.e., only processors that did not produce an error will set INFO(1) to –1 and INFO(2) to the processor having the smallest error code.

The behaviour is slightly different for INFOG(1) and INFOG(2): in the previous example, all processors would return with INFOG(1)=–8 and INFOG(2)=1000.

The possible error codes returned in INFO(1) (and INFOG(1)) have the following meaning:

–1 An error occurred on processor INFO(2).

–2 NZ is out of range. INFO(2)=NZ.

–3 MUMPS was called with an invalid value for JOB. This may happen for example if the analysis (JOB=1) was not performed before the factorization (JOB=2), or the factorization was not performed before the solve (JOB=3). See item for JOB in Section 3. This error also occurs if JOB does not contain the same value on all processes on entry to MUMPS.

–4 Error in user-provided permutation array PERM_IN in position INFO(2). This error occurs on the host only.

–5 Problem of REAL workspace allocation of size INFO(2) during analysis.

–6 Matrix is singular in structure.

–7 Problem of INTEGER workspace allocation of size INFO(2) during analysis.

–8 MAXIS too small for factorization. This may happen, for example, if numerical pivoting leads to significantly more fill-in than was predicted by the analysis. The user should increase the value of ICNTL(14) or the value of MAXIS before entering the factorization (JOB=2).

–9 MAXS too small for factorization. The user should increase the value of ICNTL(14) or MAXS before entering the factorization (JOB=2).

–10 Numerically singular matrix.

–11 MAXS too small for solution. See error INFO(1)=–9.

–12 MAXS too small for iterative refinement. See error INFO(1)=–9.

–13 Error in a Fortran ALLOCATE statement. INFO(2) contains the size that the package requested.

–14 MAXIS too small for solution. See error INFO(1)=–8.

–15 MAXIS too small for iterative refinement and/or error analysis. See error INFO(1)=–8.

–16 N is out of range. INFO(2)=N.

–17 The internal send buffer that was allocated dynamically by MUMPS on the processor is too small. The user should increase the value of ICNTL(14) before entering the analysis (JOB=1).

–18 MAXIS too small to process root node. See error INFO(1)=–8.

–19 MAXS too small to process root node. See error INFO(1)=–9.

–20 The internal reception buffer that was allocated dynamically by MUMPS on the processor is too small. INFO(2) holds the minimum size of the reception buffer required (in bytes). The user should increase the value of ICNTL(14) before entering the analysis (JOB=1).

–21 Incompatible values of PAR=0 and NPROCS=1. INFO(2)=NPROCS. Running MUMPS in host-node mode (the host is not a slave processor itself) requires at least two processors. The user should either set PAR to 1 or increase the number of processors.

–22 A pointer array is provided by the user that is either

- not associated, or
- has an insufficient size, or
- is associated and should not be associated (for example, RHS on non-host processors).

INFO(2) points to the pointer array having the wrong format in the table below:

INFO(2)	array
1	IRN or ELTPTR
2	JCN or ELTVAR
3	PERM_IN
4	A or A_ELT
5	ROWSCA
6	COLSCA
7	RHS
8	LISTVAR_SCHUR
9	SCHUR

–23 MPI was not initialized by the user prior to a call to MUMPS with JOB=–1.

–24 NELT is out of range. INFO(2)=NELT.

–25 A problem has occured in the initialization of the BLACS. This may be because you are using a vendor's BLACS. Try using a BLACS version from netlib instead.

A positive value of INFO(1) is associated with a warning message which will be output on unit ICNTL(2).

+1 Index (in IRN or JCN) out of range. Action taken by subroutine is to ignore any such entries and continue. INFO(2) is set to the number of faulty entries. Details of the first ten are printed on unit ICNTL(2).

+2 During error analysis the max-norm of the computed solution was found to be zero.

+8 Warning return from the iterative refinement routine. More than ICNTL(10) iterations are required.

+ Combinations of the above warnings will correspond to summing the constituent warnings.

8 Calling MUMPS from C

MUMPS is a Fortran 90 library, designed to be used from Fortran 90 rather than C. However a basic C interface is provided that allows users to call MUMPS directly from C programs. Similarly to the Fortran 90 interface, the C interface uses a structure whose components match those in the MUMPS structure for Fortran (Figure 1). Thus the description of the parameters in Sections 4and 5 applies. Figure 2 shows the C structure `[SDCZ]MUMPS_STRUC_C`. This structure is defined in the include file `[sdcz]mumps_c.h` and there is one main routine per available precision with the following prototype:

```
void [sdcz]mumps_c(MUMPS_STRUC_C * idptr);
```

An example of calling MUMPS from C for a complex assembled problem is given in Section 9.3. The following subsections discuss some technical issues that a user should be aware of before using the C interface to MUMPS.

In the following, we suppose that `id` has been declared of type `[SDCZ]MUMPS_STRUC_C`.

8.1 Array indices

Arrays in C start at index 0 whereas they normally start at 1 in Fortran. Therefore, care must be taken when providing arrays to the C structure. For example, the row indices of the matrix A, stored in `IRN(1:NZ)` in the Fortran version should be stored in `irn[0:nz-1]` in the C version. (Note that the contents of `irn` itself is unchanged with values between 1 and N.) One solution to deal with this is to define macros:

```
typedef struct
   {
    int sym, par, job;
    int comm_fortran; /* Fortran communicator */
    int icntl[40];
    real cntl[5];
    int n;
    /* Assembled entry */
    int nz; int *irn; int *jcn; real/complex *a;
    /* Distributed entry */
    int nz_loc; int *irn_loc; int *jcn_loc; real/complex *a_loc;
    /* Element entry */
    int nelt; int *eltptr; int *eltvar; real/complex *a_elt;
    /* Ordering, if given by user */
    int *perm_in;
    /* Scaling (input only in this version) */
    real/complex *colsca; real/complex *rowsca;
    /* Output data and statistics */
    real/complex *rhs;
    int info[40],infog[40];
    real rinfo[20], rinfog[20];
    int *sym_perm, *uns_perm;
    /* Null space (not maintained) */
    int deficiency; real/complex * nullspace; int * mapping;
    /* Schur */       int size_schur; int *listvar_schur; real/complex *schur;
    /* Internal parameters */
    int instance_number;
   } [SDCZ]MUMPS_STRUC_C;
```

Figure 2: Definition of the C structure [SDCZ]MUMPS_STRUC_C. **real/complex** is used for data that can be either real or complex, **real** for data that stays real (float or double) in the complex version.

```
#define ICNTL( i ) icntl[ (i) - 1 ]
#define A( i ) a[ (i) -1 ]
#define IRN( i ) irn[ (i) -1 ]
...
```

and then use the uppercase notation with parenthesis (instead of lowercase/brackets). In that case, the notation `id.IRN(I)`, where `I` is in { 1, 2, ... NZ} can be used instead of `id.irn[I-1]`; this notation then matches exactly with the description in Sections 4 and 5, where arrays are supposed to start at 1.

This can be slightly more confusing for element matrix input (see Section 4.5), where some arrays are used to index other arrays. For instance, the first value in `eltptr`, `eltptr[0]`, pointing into the list of variables of the first element in `eltvar`, should be equal to 1. Effectively, using the notation above, the list of variables for element $j = 1$ starts at location `ELTVAR(ELTPTR(j)) = ELTVAR(eltptr[j-1]) = eltvar[eltptr[j-1]-1]`.

8.2 Issues related to the C and Fortran communicators

In general, C and Fortran communicators have a different datatype and are not directly compatible. For the C interface, `MUMPS` requires a Fortran communicator to be provided in `id.comm_fortran`. If, however, this field is initialized to the special value -987654, the Fortran communicator `MPI_COMM_WORLD` is used by default. If you need to call `MUMPS` based on a smaller number of processors defined by a C subcommunicator, then you should convert your C communicator to a Fortran one. This has not been included in `MUMPS` because it is dependent on the `MPI` implementation and thus not portable. For `MPI2`, you may just do

```
id.comm_fortran = (F_INT) MPI_Comm_c2f(comm_c);
```

(Note that `F_INT` is defined in `[sdcz]mumps_c.h` and normally is an int.) For MPI implementations where the Fortran and the C communicators have the same integer representation

```
id.comm_fortran = (F_INT) MPI_Comm_c2f(comm_c);
```

should work.

For MPICH, check if `id.comm_fortran = MPIR_FromPointer(comm_c)` gives the expected result.

8.3 Fortran I/O

Diagnostic, warning and error messages (controlled by `ICNTL(1:4)` / `icntl[0..3]`) are based on Fortran file units. Use the value 6 for the Fortran unit 6 which corresponds to `stdout`. For a more general usage with specific file names from C, passing a C file handler is not currently possible. One solution would be to use a Fortran subroutine along the lines of the model below:

```
SUBROUTINE OPENFILE( UNIT, NAME )
INTEGER UNIT
CHARACTER*(*) NAME
OPEN(UNIT, file=NAME)
RETURN
END
```

and have (in the C user code) a statement like

```
openfile_( &mumps_par.ICNTL(1), name, name_length_byval)
```

(or slightly different depending on the C-Fortran calling conventions); something similar could be done to close the file.

8.4 Runtime libraries

The Fortran 90 runtime library corresponding to the compiler used to compile `MUMPS` is required at the link stage. One way to provide it is to perform the link phase with the Fortran compiler (instead of the C compiler or `ld`).

8.5 Integer, real and complex datatypes in C and Fortran

We assume that the `int`, `float` and `double` types are compatible with the Fortran `INTEGER`, `REAL` and `DOUBLE PRECISION` datatypes. If this was not the case, the files `[dscz]mumps_prec.h` or Makefiles would need to be modified accordingly.

Since not all C compilers define the `complex` datatype (this only appeared in the C99 standard), we define the following, compatible with the Fortran `COMPLEX` and `DOUBLE COMPLEX` types:

`typedef struct {float r,i;} mumps_complex;` for simple precision (`cmumps`), and

`typedef struct {double r,i;} mumps_double_complex;` for double precision (`zmumps`).

Types for complex data from the user program should be compatible with those above.

8.6 Sequential version

The C interface to `MUMPS` is compatible with the sequential version; see Section 2.9.

9 Examples of use of `MUMPS`

9.1 An assembled problem

An example program illustrating a possible use of `MUMPS` on assembled `DOUBLE PRECISION` problems is given Figure 3. Two files must be included in the program: `mpif.h` for MPI and `mumps_struc.h` for `MUMPS`. The file `mumps_root.h` must also be available because it is included in `mumps_struc.h`. The initialization and termination of MPI are performed in the user program via the calls to `MPI_INIT` and `MPI_FINALIZE`.

The `MUMPS` package is initialized by calling `MUMPS` with JOB=–1, the problem is read in by the host (in the components N, NZ, IRN, JCN, A, and RHS), and the solution is computed in RHS with a call on all processors to `MUMPS` with JOB=6. Finally, a call to `MUMPS` with JOB=–2 is performed to deallocate the data structures used by the instance of the package.

Thus for the assembled 5×5 matrix and right-hand side

$$\begin{pmatrix} 2 & 3 & 4 & & \\ 3 & & -3 & & 6 \\ & -1 & 1 & 2 & \\ & & 2 & & \\ & 4 & & & 1 \end{pmatrix}, \qquad \begin{pmatrix} 20 \\ 24 \\ 9 \\ 6 \\ 13 \end{pmatrix}$$

we could have as input

```
5             : N
12            : NZ
1 2 3.0
2 3 -3.0
4 3 2.0
5 5 1.0
2 1 3.0
1 1 2.0
5 2 4.0
3 4 2.0
2 5 6.0
3 2 -1.0
1 3 4.0
3 3 1.0        : A
20.0
24.0
9.0
6.0
```

```
      PROGRAM MUMPS_EXAMPLE
      INCLUDE 'mpif.h'
      INCLUDE 'dmumps_struc.h'
      TYPE (DMUMPS_STRUC) id
      INTEGER IERR, I
      CALL MPI_INIT(IERR)
C  Define a communicator for the package
      id%COMM = MPI_COMM_WORLD
C  Ask for unsymmetric code
      id%SYM = 0
C  Host working
      id%PAR = 1
C  Initialize an instance of the package
      id%JOB = -1
      CALL DMUMPS(id)
C  Define problem on the host (processor 0)
      IF ( id%MYID .eq. 0 ) THEN
        READ(5,*) id%N
        READ(5,*) id%NZ
        ALLOCATE( id%IRN ( id%NZ ) )
        ALLOCATE( id%JCN ( id%NZ ) )
        ALLOCATE( id%A( id%NZ ) )
        ALLOCATE( id%RHS ( id%N  ) )
        READ(5,*) ( id%IRN(I) ,I=1, id%NZ )
        READ(5,*) ( id%JCN(I) ,I=1, id%NZ )
        READ(5,*) ( id%A(I),I=1, id%NZ )
        READ(5,*) ( id%RHS(I) ,I=1, id%N  )
      END IF
C  Call package for solution
      id%JOB = 6
      CALL DMUMPS(id)
C  Solution has been assembled on the host
      IF ( id%MYID .eq. 0 ) THEN
        WRITE( 6, * ) ' Solution is ',(id%RHS(I),I=1,id%N)
      END IF
C  Deallocate user data
      IF ( id%MYID .eq. 0 )THEN
        DEALLOCATE( id%IRN )
        DEALLOCATE( id%JCN )
        DEALLOCATE( id%A   )
        DEALLOCATE( id%RHS )
      END IF
C  Destroy the instance (deallocate internal data structures)
      id%JOB = -2
      CALL DMUMPS(id)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 3: Example program using MUMPS on an assembled DOUBLE PRECISION problem

```
13.0              :RHS
```

and we obtain the solution RHS(i) = i, i = 1, . . . , 5.

9.2 An elemental problem

An example of a driver to use `MUMPS` for element `DOUBLE PRECISION` problems is given in Figure 4. The calling sequence is similar to that for the assembled problem in Section 9.1 but now the host reads the problem in components N, NELT, ELTPTR, ELTVAR, A_ELT, and RHS. Note that for elemental problems ICNTL(5) must be set to 1 and that elemental matrices always have a symmetric structure. For the two-element matrix and right-hand side

$$\begin{matrix}1\\2\\3\end{matrix}\begin{pmatrix}-1 & 2 & 3\\ 2 & 1 & 1\\ 1 & 1 & 1\end{pmatrix}, \qquad \begin{matrix}3\\4\\5\end{matrix}\begin{pmatrix}2 & -1 & 3\\ 1 & 2 & -1\\ 3 & 2 & 1\end{pmatrix}, \qquad \begin{pmatrix}12\\7\\23\\6\\22\end{pmatrix}$$

we could have as input

```
5
2
6
18
1 4 7
1 2 3 3 4 5
-1.0 2.0 1.0 2.0 1.0 1.0 3.0 1.0 1.0 2.0 1.0 3.0 -1.0 2.0 2.0 3.0 -1.0 1.0
12.0 7.0 23.0 6.0 22.0
```

and we obtain the solution RHS(i) = i, i = 1, . . . , 5.

9.3 An example of calling MUMPS from C

An example of a driver to use `MUMPS` from C is given in Figure 5.

Acknowledgements

`MUMPS` has been partially supported by the ESPRIT IV Project PARASOL, and by CERFACS, ENSEEIHT-IRIT, INRIA Rhone-Alpes, LBNL-NERSC, Parallab and the Rutherford-Appleton Laboratory.

The functionalities related to rank-revealing on the root of the multifrontal tree were implemented by M. Tůma [3] while he was at CERFACS and are not anymore maintained.

We are also grateful to Caroline Bousquet, Christophe Daniel, Abdou Guermouche, Chiara Puglisi, Séphane Pralet, Grégoire Richard, and Christophe Vömel who have been contributed to this work.

[3]tuma@uivt.cas.cz, Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod vodárenskou, věží 2, 182 07 Praha 8, Czech Republic.

```
      PROGRAM MUMPS_EXAMPLE
      INCLUDE 'mpif.h'
      INCLUDE 'dmumps_struc.h'
      TYPE (DMUMPS_STRUC) id
      INTEGER IERR, LELTVAR, NA_ELT
      CALL MPI_INIT(IERR)
C  Define a communicator for the package
      id%COMM = MPI_COMM_WORLD
C  Ask for unsymmetric code
      id%SYM = 0
C  Host working
      id%PAR = 1
C  Initialize an instance of the package
      id%JOB = -1
      CALL DMUMPS(id)
C  Define the problem on the host (processor 0)
      IF ( id%MYID .eq. 0 ) THEN
        READ(5,*) id%N
        READ(5,*) id%NELT
        READ(5,*) LELTVAR
        READ(5,*) NA_ELT
        ALLOCATE( id%ELTPTR ( id%NELT+1 ) )
        ALLOCATE( id%ELTVAR ( LELTVAR ) )
        ALLOCATE( id%A_ELT( NA_ELT ) )
        ALLOCATE( id%RHS ( id%N  ) )
        READ(5,*) ( id%ELTPTR(I) ,I=1, id%NELT+1 )
        READ(5,*) ( id%ELTVAR(I) ,I=1, LELTVAR )
        READ(5,*) ( id%A_ELT(I),I=1, NA_ELT )
        READ(5,*) ( id%RHS(I) ,I=1, id%N  )
      END IF
C  Specify element entry
      id%ICNTL(5) = 1
C  Call package for solution
      id%JOB = 6
      CALL DMUMPS(id)
C  Solution has been assembled on the host
      IF ( id%MYID .eq. 0 ) THEN
        WRITE( 6, * ) ' Solution is ',(id%RHS(I),I=1,id%N)
      END IF
C  Deallocate user data
      DEALLOCATE( id%ELTPTR )
      DEALLOCATE( id%ELTVAR )
      DEALLOCATE( id%A_ELT )
      DEALLOCATE( id%RHS )
C  Destroy the instance (deallocate internal data structures)
      id%JOB = -2
      CALL DMUMPS(id)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 4: Example program using MUMPS on an element DOUBLE PRECISION problem.

```
/* Example program using the C interface to the
 * double precision version of MUMPS, dmumps_c.
 * We solve the system A x = RHS with
 *   A = diag(1 2) and RHS = [1 4]ˆT
 * Solution is [1 2]ˆT */
#include <stdio.h>
#include "mpi.h"
#include "dmumps_c.h"
#define JOB_INIT -1
#define JOB_END -2
#define USE_COMM_WORLD -987654
int main(int argc, char ** argv) {
  DMUMPS_STRUC_C id;
  int n = 2;
  int nz = 2;
  int irn[] = {1,2};
  int jcn[] = {1,2};
  double a[2];
  double rhs[2];

  int myid, ierr;
  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  /* Define A and rhs */
  rhs[0]=1.0;rhs[1]=4.0;
  a[0]=1.0;a[1]=2.0;

   /* Initialize a MUMPS instance. Use MPI_COMM_WORLD.  */
  id.job=JOB_INIT; id.par=1; id.sym=0;id.comm_fortran=USE_COMM_WORLD;
  dmumps_c(&id);
  /* Define the problem on the host */
  if (myid == 0) {
    id.n = n; id.nz =nz; id.irn=irn; id.jcn=jcn;
    id.a = a; id.rhs = rhs;
  }
#define ICNTL(I) icntl[(I)-1] /* macro s.t. indices match documentation */
/* No outputs */
  id.ICNTL(1)=-1; id.ICNTL(2)=-1; id.ICNTL(3)=-1; id.ICNTL(4)=0;
/* Call the MUMPS package. */
  id.job=6;
  dmumps_c(&id);
  id.job=JOB_END; dmumps_c(&id); /* Terminate instance */
  if (myid == 0) {
    printf("Solution is : (%8.2f  %8.2f)\n", rhs[0],rhs[1]);
  }
  return 0;
}
```

Figure 5: Example program using MUMPS from C on an assembled problem.

References

[1] P. R. Amestoy. Recent progress in parallel multifrontal solvers for unsymmetric sparse matrices. In *Proceedings of the 15th World Congress on Scientific Computation, Modelling and Applied Mathematics, IMACS 97, Berlin*, 1997.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17:886–905, 1996.

[3] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *Int. J. of Supercomputer Applics.*, 3:41–59, 1989.

[4] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[5] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal solvers within the PARASOL environment. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 7–11, Berlin, 1998. Springer-Verlag.

[6] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Parallélisation de la factorisation LU de matrices creuses non-symétriques pour des architectures à mémoire distribuée. *Calculateurs Parallèles Réseaux et Systèmes Répartis*, 10(5):509–520, 1998.

[7] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[8] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10:165–190, 1989.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[10] A. R. Curtis and J. K. Reid. On the automatic scaling of matrices for Gaussian elimination. *J. Inst. Maths. Applics.*, 10:118–124, 1972.

[11] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. Algorithm 679. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[12] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. Algorithm 679. A set of Level 3 Basic Linear Algebra Subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software*, 16:18–28, 1990.

[13] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):889–901, 1999.

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

[15] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[16] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[17] G. Karypis and V. Kumar. METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, September 1998.

[18] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.