

MUltifrontal Massively Parallel Solver (MUMPS 4.10.0) Users' guide *

May 10, 2011

Abstract

This document describes the Fortran 90 and C user interfaces to MUMPS 4.10.0. We describe in detail the data structures, parameters, calling sequences, and error diagnostics. Basic example programs using MUMPS are also provided.

*Information on how to obtain updated copies of MUMPS can be obtained from the Web pages <http://mumps.enseeiht.fr/> and <http://graal.ens-lyon.fr/MUMPS/>

Contents

1	Introduction	4
2	Main functionalities of MUMPS 4.10.0	5
2.1	Input matrix structure	5
2.2	Analysis/Preprocessing	6
2.3	Post-processing facilities	7
2.4	Solving the transposed system	7
2.5	Arithmetic versions	7
2.6	The working host processor	7
2.7	Sequential version	8
2.8	Shared memory version	8
2.9	Out-of-core facility	8
2.10	Determinant	8
2.11	Computing entries of A^{-1}	8
2.12	Reduce/condense a problem on an interface (Schur complement, reduced/condensed RHS)	9
3	User interface and available routines	10
4	Input and output parameters	13
4.1	Version number	13
4.2	Control of the three main phases: Analysis, Factorization, Solve	13
4.3	Control of parallelism	15
4.4	Matrix type	15
4.5	Centralized assembled matrix input: ICNTL(5)=0 and ICNTL(18)=0	15
4.6	Element matrix input: ICNTL(5)=1 and ICNTL(18)=0	16
4.7	Distributed assembled matrix input: ICNTL(5)=0 and ICNTL(18)≠0	16
4.8	Scaling: ICNTL(8)	17
4.9	Given ordering: ICNTL(7)=1	17
4.10	Schur complement with reduced (or condensed) right-hand side: ICNTL(19), ICNTL(26)	17
4.11	Out-of-core (ICNTL(22)≠ 0)	21
4.12	Workspace parameters	21
4.13	Right-hand side and solution vectors/matrices	21
4.14	Writing a matrix to a file	23
5	Control parameters	23
5.1	Integer control parameters	23
5.2	Real/complex control parameters	31
5.3	Compatibility between options	32
6	Information parameters	34
6.1	Information local to each processor	34
6.2	Information available on all processors	36
7	Error diagnostics	38
8	Calling MUMPS from C	41
8.1	Array indices	41
8.2	Issues related to the C and Fortran communicators	41
8.3	Fortran I/O	43
8.4	Runtime libraries	43
8.5	Integer, real and complex datatypes in C and Fortran	43
8.6	Sequential version	43
9	Scilab and MATLAB/Octave interfaces	43

10 Examples of use of MUMPS	45
10.1 An assembled problem	45
10.2 An elemental problem	47
10.3 An example of calling MUMPS from C	49
11 Notes on MUMPS distribution	51

1 Introduction

MUMPS (“MULTifrontal Massively Parallel Solver”) is a package for solving systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where \mathbf{A} is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric. MUMPS implements a direct method based on a multifrontal approach which performs a direct factorization

$$\mathbf{A} = \mathbf{LU} \tag{1}$$

where \mathbf{L} is a lower triangular matrix and \mathbf{U} an upper triangular matrix. If the matrix is symmetric then the factorization

$$\mathbf{A} = \mathbf{LDL}^T \tag{2}$$

where \mathbf{D} is block diagonal matrix with blocks of order 1 or 2 on the diagonal is performed. We refer the reader to the papers [5, 6, 9, 20, 21, 25, 24, 11] 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, out-of-core capability, parallel analysis, detection of null pivots, basic estimate of rank deficiency and null space basis for symmetric matrices, and computation of a Schur complement matrix. MUMPS offers several built-in ordering algorithms, a tight interface to some external ordering packages such as PORD [31], SCOTCH [28] or METIS [26] (strongly recommended), and the possibility for the user to input a given ordering. Finally, MUMPS is available in various arithmetics (real or complex, single or double precision). A parallel analysis and an out-of-core functionality are also available. Most recent experimental functionalities involve the computation of the determinant, the computation of the entries in the inverse of \mathbf{A} and exploiting sparsity of the right-hand sides to reduce the amount of floating-point operations and accesses to the factor matrices.

The software is mainly written in Fortran 90 although a C interface is available (see Section 8). Scilab and MATLAB/Octave interfaces are also available in the case of sequential executions. The parallel version of MUMPS requires MPI [33] for message passing and makes use of the BLAS [15, 16], BLACS, and ScaLAPACK [13] libraries. The sequential version only relies on BLAS.

MUMPS is downloaded from the web site almost four times a day on average and has been run on very many machines, compilers and operating systems, although our experience is really only with UNIX-based systems. We have tested it extensively on parallel computers from SGI, Cray, and IBM and on clusters of workstations.

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

1. Analysis.

During analysis, preprocessing (see Section 2.2) including an ordering based on the symmetrized pattern $\mathbf{A} + \mathbf{A}^T$ and a symbolic factorization is performed. Both parallel and sequential implementation of the analysis phase is available. A mapping of the multifrontal computational graph is then computed and used to estimate the number of operations and memory necessary for factorization and solution. Let \mathbf{A}_{pre} denotes the preprocessed matrix (further defined in Section 2.2).

2. Factorization.

During factorization a direct factorization $\mathbf{A}_{\text{pre}} = \mathbf{LU}$ or $\mathbf{A}_{\text{pre}} = \mathbf{LDL}^T$ depending on the symmetry of the preprocessed matrix is computed. The original matrix is first distributed (or redistributed) onto the processors depending on the mapping of the dependency graph of factorization, the so called **elimination tree** [27]. The numerical factorization is then a sequence of dense factorization on so called **frontal matrices**. In addition to standard threshold pivoting and two-by-two pivoting (not so standard in distributed memory codes) there is an option to perform static pivoting. The elimination tree also expresses independency between tasks and enables multiple fronts to be processed simultaneously. This approach is called **multifrontal approach**. After the factorization, the factor matrices are kept distributed (in core memory or on disk); they will be used at the solution phase.

3. Solution.

The solution of $\mathbf{LU}\mathbf{x}_{\text{pre}} = \mathbf{b}_{\text{pre}}$ or $\mathbf{LDL}^T\mathbf{x}_{\text{pre}} = \mathbf{b}_{\text{pre}}$ where \mathbf{x}_{pre} and \mathbf{b}_{pre} are respectively the transformed solution \mathbf{x} and right-hand side \mathbf{b} associated to the preprocessed matrix \mathbf{A}_{pre} , is obtained through a **forward** elimination step

$$\mathbf{L}\mathbf{y} = \mathbf{b}_{\text{pre}} \text{ or } \mathbf{LD}\mathbf{y} = \mathbf{b}_{\text{pre}} , \quad (3)$$

followed by a **backward** elimination step

$$\mathbf{U}\mathbf{x}_{\text{pre}} = \mathbf{y} \text{ or } \mathbf{L}^T\mathbf{x}_{\text{pre}} = \mathbf{y} . \quad (4)$$

The right-hand side b is first preprocessed and then broadcasted from the host to the working processors. Sparse right-hand sides might be used to limit the volume of data exchange during this step. A forward elimination (Equation 3) and a backward substitution (Equation 4) are then performed using the (distributed) factors computing during factorization to obtain \mathbf{x}_{pre} . The solution \mathbf{x}_{pre} is finally postprocessed to obtain the solution \mathbf{x} of the original system $\mathbf{A}\mathbf{x} = \mathbf{b}$, where \mathbf{x} is either assembled on the host or kept distributed on the working processors. Iterative refinement and backward error analysis are also postprocessing options of the solution phase.

Each of these phases can be called separately and several instances of MUMPS can be handled simultaneously. MUMPS allows the host processor to participate to the factorization and solve phases, just like any other processor (see Section 2.6).

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 during the analysis phase.

2 Main functionalities of MUMPS 4.10.0

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 `mumps_par` of type `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 for inputting the matrix. The selection is controlled by the parameters `ICNTL(5)` and `ICNTL(18)`.

The input matrix can be supplied in *elemental format* and must then be input centrally on the host (`ICNTL(5)=1` and `ICNTL(18)=0`). For full details see Section 4.6. Otherwise, it can be supplied in *assembled format* in coordinate form (`ICNTL(5)=0`), and, in this case, there are several possibilities (see Sections 4.5 and 4.7):

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 (recommended option for distributed entry).

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 Analysis/Preprocessing

A range of symmetric orderings to preserve sparsity is available during the analysis phase. In addition to the symmetric orderings, the package offers pre-processing facilities: permuting to zero-free diagonal and prescaling. When all preprocessing options are activated, the preprocessed matrix \mathbf{A}_{pre} that will be effectively factored is :

$$\mathbf{A}_{\text{pre}} = \mathbf{P} \mathbf{D}_r \mathbf{A} \mathbf{Q}_c \mathbf{D}_c \mathbf{P}^T, \quad (5)$$

where \mathbf{P} is a permutation matrix applied symmetrically, \mathbf{Q}_c is a (column) permutation and \mathbf{D}_r and \mathbf{D}_c are diagonal matrices for (respectively row and column) scaling. Note that when the matrix is symmetric, preprocessing is designed to preserved symmetry.

Preprocessing highly influences the performance (memory and time) of the factorization and solution steps. The default values correspond to an automatic setting performed by the package which depends on the ordering packages installed, the type of the matrix (symmetric or unsymmetric), the size of the matrix and the number of processors available. We thus strongly recommend the user to install all ordering packages to offer maximum choice to the automatic decision process.

- Symmetric permutation : \mathbf{P}

The symmetric permutation can be computed either sequentially, or in parallel. The `ICNTL(28)` parameter is responsible for setting the strategy.

In the case where the symmetric permutation is computed sequentially, the ordering method is set by the `ICNTL(7)` parameter which offers a range of ordering options including the approximate minimum degree ordering (AMD, [4]), an approximate minimum degree ordering with automatic quasi-dense row detection (QAMD, [3]), an approximate minimum fill-in ordering (AMF), an ordering where bottom-up strategies are used to build separators by Jürgen Schulze from University of Paderborn (PORD, [31]), the SCOTCH package [28], and the METIS package from Univ. of Minnesota [26]. A user-supplied ordering can also be provided and the pivot order must be set by the user on the host in `PERM_IN` (see Section 4.9).

In the case where the symmetric permutation is computed in parallel, the ordering method is set by the `ICNTL(29)`. One of the PT-SCOTCH and ParMetis parallel ordering tools can used in this case.

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

- Permutations to a zero-free diagonal : \mathbf{Q}_c

Controlled by `ICNTL(6)`, this permutation is recommended for very unsymmetric matrices to reduce fill-in and arithmetic cost, see [17, 18]. For symmetric matrices this permutation can also be used to constrain the symmetric permutation (see also `ICNTL(12)` option). Furthermore, when numerical values are provided on entry to the analysis phase, `ICNTL(6)` may also build scaling vectors during the analysis, that will be either used or discarded depending on the scaling option `ICNTL(8)`.

- Row and Column scalings : \mathbf{D}_r and \mathbf{D}_c

Controlled by `ICNTL(8)`, this preprocessing improves the numerical accuracy and makes all estimations performed during analysis more reliable. A range of classical scalings are provided and can be automatically performed within the package (see Section 4.8), either during the analysis phase or at the beginning of the factorization phase.

Furthermore, preprocessing strategies for symmetric indefinite matrices, as described in [19], can be applied and also lead to scaling arrays; they are controlled by `ICNTL(12)`.

2.3 Post-processing facilities

It has been shown [12] 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 [12]. 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

$$\frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{b}| + |\mathbf{A}| |\bar{\mathbf{x}}|)_i} \quad (6)$$

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

$$\frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{A}| |\bar{\mathbf{x}}|)_i + \|\mathbf{A}_i\|_\infty \|\bar{\mathbf{x}}\|_\infty} \quad (7)$$

is used instead, where \mathbf{A}_i is row i of \mathbf{A} . The largest scaled residual (6) is returned in `RINFOG(7)` and the largest scaled residual (7) 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

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

and $\delta\mathbf{b}_i \leq \max(\text{RINFOG}(7)|\mathbf{b}|_i, \text{RINFOG}(8)\|\mathbf{A}_i\|_\infty \|\bar{\mathbf{x}}\|_\infty)$. Note that $\delta\mathbf{A}$ respects the sparsity of \mathbf{A} in the sense that $\delta\mathbf{A}_{ij}$ is zero for structural zeros in \mathbf{A} , i.e., when $\mathbf{A}_{ij}=0$. An upper bound for the error in the solution is returned in `RINFOG(9)`. Finally condition numbers $cond_1$ and $cond_2$ for the linear system (not just the matrix) are returned in `RINFOG(10)` and `RINFOG(11)`, respectively, and

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \text{RINFOG}(9) = \text{RINFOG}(7) \times cond_1 + \text{RINFOG}(8) \times cond_2.$$

2.4 Solving the transposed system

Given a sparse matrix \mathbf{A} , the system $\mathbf{A}\mathbf{X} = \mathbf{B}$ or $\mathbf{A}^T\mathbf{X} = \mathbf{B}$ can be solved during the solve stage, where \mathbf{A} is square of order n and \mathbf{X} and \mathbf{B} are of order n by $nrhs$. This is controlled by `ICNTL(9)`.

2.5 Arithmetic versions

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

To compile all or any particular version, please refer to the root README of the MUMPS sources.

This document applies to all four arithmetics. 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`,

2.6 The working host processor

The analysis phase is performed on the host processor. This processor is the one with rank 0 in the communicator provided to MUMPS. By setting the variable `PAR` to 1 (see Section 4.3), MUMPS allows the host to participate in computations during the factorization and solve phases, just like any other processor. This allows MUMPS to run on a single processor and prevents the host processor being idle during the factorization and solve phases (as would be the case for `PAR=0`). We thus generally recommend using 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 the storage related to the initial matrix on the host.

2.7 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 needs 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 that provides all external references needed by MUMPS for a sequential environment. MUMPS can thus be used in a simple sequential program, ignoring everything related to parallelism or 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 4.3) and that the calling program should not make use of MPI.

2.8 Shared memory version

On networks of SMP nodes (multiprocessor nodes with a shared memory) or on multicore-based machines, a parallel shared memory BLAS library (also called multithread BLAS) is often available. Using shared memory or threaded 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.

2.9 Out-of-core facility

Controlled by `ICNTL(22)`, a preliminary out-of-core facility is available in both sequential and parallel environments. In this version only the factors are written to disk during the factorization phase and will be read each time a solution phase is requested. Our experience is that on a reasonably small number of processors this can significantly reduce the memory requirement while not increasing much the factorization time. The extra cost of the out-of-core feature is thus mainly during the solve phase, where factors have to be read from disk for both the forward elimination and the backward substitution.

2.10 Determinant

Controlled by `ICNTL(33)`, MUMPS has an option to compute the determinant of the matrix provided on entry. It is available for symmetric and unsymmetric matrices for all arithmetics (single, double, real, complex), and for all matrix input formats.

If $A = LU$ (unsymmetric matrices), then $\det(A) = \det(L) \times \det(U) = \prod_{i=1}^n U_{ii}$, where n is the order of the matrix A . If $A = LDL^t$ (symmetric matrices), then $\det(A) = \prod_{i=1}^n D_{ii}$. The sign of the determinant is maintained by keeping track of all internal permutations. Scaling arrays are taken into account too, in case the matrix is scaled. To avoid overflows and guarantee an accurate computation, the mantissa and exponent are computed separately and renormalized when needed.

The determinant is only computed when requested by the user. For more information, see `ICNTL(33)`. If the user is only interested in the determinant, he/she may tell MUMPS that the factor matrices can be discarded (see `ICNTL(31)`), significantly reducing the storage requirements.

2.11 Computing entries of A^{-1}

Several applications require the explicit computation of selected entries of the inverse of large sparse matrices. In most cases, many entries are requested, for example all diagonal entries. To compute column j of the inverse, the equation $Ax = e_j$ can be used, where e_j is the j th column of the identity matrix. One can obtain major savings if the structural zeros of e_j are exploited or if only few entries of the j th column

are requested [32, 2]. If we have an LU factorization of A , a_{ij}^{-1} , the (i, j) entry of A^{-1} , is obtained by solving successively the two triangular systems:

$$y = L^{-1}e_j \quad (8)$$

$$a_{ij}^{-1} = (U^{-1}y)_i \quad (9)$$

MUMPS provides a functionality, controlled by `ICNTL(30)`, to compute a set of entries of A^{-1} , while avoiding most of the computations on explicit zeros in Equations (8) and (9). The list of entries of A^{-1} to be computed and the memory for those entries should be provided as a sparse right-hand side, see the descriptions of `ICNTL(30)` in Section 5, and of `IRHS_PTR`, `IRHS_SPARSE`, and `RHS_SPARSE` in Section 4.13. In a parallel environment there are quite a lot of opportunities for improvements which are the topic of on going research activities but are not yet included in this release.

2.12 Reduce/condense a problem on an interface (Schur complement and reduced/condensed right-hand side)

A Schur complement matrix (centralized or provided as 2D block cyclic matrix) can be returned to the user (see `ICNTL(19)`, `ICNTL(26)` and Section 4.10). 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 (`ICNTL(26)=0`) and also to handle a reduced/condensed right-hand-side (`ICNTL(26)=1,2`) as described in the following discussion.

Let us consider a partitioned matrix (here with an unsymmetric matrix) where the variables of $A_{2,2}$, specified by the user, correspond to the Schur variables and on which a partial factorization has been performed. In the following, and only for the sake of clearness we have ordered last all variables belonging to the Schur.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & I \end{pmatrix} \begin{pmatrix} U_{1,1} & U_{1,2} \\ 0 & S \end{pmatrix} \quad (10)$$

Thus the Schur complement, as returned by MUMPS, is such that $S = A_{2,2} - A_{2,1}A_{1,1}^{-1}A_{1,2}$.

`ICNTL(26)` can then be used during the solution phase to describe how this partial factorization can be used to solve $Ax = b$:

- **Compute a partial solution**:

If `ICNTL(26)=0` then the solve is performed on the internal problem:

$$A_{1,1}x_1 = b_1.$$

Entries in the right-hand side corresponding to indices from the Schur matrix need not be set on entry and they are explicitly set to zero on output.

- **Solve the complete system in three steps**:

$$\begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & I \end{pmatrix} \begin{pmatrix} U_{1,1} & U_{1,2} \\ 0 & S \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad (11)$$

1. **Reduction/condensation phase**:

One can compute with `ICNTL(26)=1`, the intermediate y vector, in which y_2 is often referred to as the reduced/condensed right-hand-side.

$$\begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & I \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad (12)$$

Then one has to solve

$$\begin{pmatrix} \mathbf{U}_{1,1} & \mathbf{U}_{1,2} \\ 0 & \mathbf{S} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \quad (13)$$

2. **Using Schur matrix** :
The Schur matrix is an output of the factorisation phase. It is the responsibility of the user to compute x_2 such that $\mathbf{S}x_2 = y_2$.
3. **Expansion phase** :
Given x_2 and y_1 , option `ICNTL(26)=2` of the solve phase can be used to compute x_1 . Note that the package uses y_1 computed (and stored in the `mumps` structure) during the first step (`ICNTL(26)=1`) and that the complete solution x is provided on output.

Note that the Schur complement could be considered as an element contribution to the interface block in a domain decomposition approach. MUMPS could then be used to solve this interface problem using the element entry functionality.

3 User interface and available routines

In the following, we use the notation `[SDCZ]MUMPS` to refer to `DMUMPS`, `SMUMPS`, `ZMUMPS` or `CMUMPS`, corresponding to the `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_struct.h` to `smumps_struct.h`, `dmumps_struct.h`, `cmumps_struct.h` or `zmumps_struct.h`.

In the Fortran 90 interface (see Section 8 for the C interface), there is a single user callable subroutine per arithmetic, called `[SDCZ]MUMPS`, that has a single parameter `mumps_par` of Fortran 90 derived datatype `[SDCZ]MUMPS_STRUC` defined in `[sdcz]mumps_struct.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_struct.h'
...
INTEGER IERR
TYPE (DMUMPS_STRUC) :: mumps_par
...
CALL MPI_INIT(IERR)      ! Not needed in purely sequential version
...
mumps_par%JOB = ...     ! Set some arguments to the package: those
mumps_par%ICNTL(3)=6    ! are components of the mumps_par structure
...
CALL DMUMPS( mumps_par )
...
CALL MPI_FINALIZE(IERR) ! Not needed in purely sequential version

```

For other arithmetics, `dmumps_struct.h` should be replaced by `smumps_struct.h`, `cmumps_struct.h`, or `zmumps_struct.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_struct.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_struct.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)
C     INTEGER SYM, PAR, JOB
C   Control parameters
C   -----
C     INTEGER ICNTL(40)
C
C     real CNTL(15)
C     INTEGER N ! Order of input matrix
C   Assembled input matrix : User interface
C   -----
C     INTEGER NZ
C
C     real/complex , DIMENSION(:), POINTER :: A
C     INTEGER, DIMENSION(:), POINTER :: IRN, JCN
C   Case of distributed matrix entry
C   -----
C     INTEGER NZ_loc
C     INTEGER, DIMENSION(:), POINTER :: IRN_loc, JCN_loc
C
C     real/complex , DIMENSION(:), POINTER :: A_loc
C   Unassembled input matrix: User interface
C   -----
C     INTEGER NELT
C     INTEGER, DIMENSION(:), POINTER :: ELTPTR, ELTVAR
C
C     real/complex , DIMENSION(:), POINTER :: A_ELTPTR
C   MPI Communicator and identifier
C   -----
C     INTEGER COMM, MYID
C   Ordering and scaling, if given by user (optional)
C   -----
C     INTEGER, DIMENSION(:), POINTER :: PERM_IN
C
C     real/complex , DIMENSION(:), POINTER :: COLSCA, ROWSCA
C INPUT/OUTPUT data : right-hand side and solution
C -----
C
C     real/complex , DIMENSION(:), POINTER :: RHS, REDRHS
C     real/complex , DIMENSION(:), POINTER :: RHS_SPARSE
C
C     INTEGER, DIMENSION(:), POINTER :: IRHS_SPARSE, IRHS_PTR
C     INTEGER NRHS, LRHS, NZ_RHS, LSOL_loc, LREDRHS
C
C     real/complex , DIMENSION(:), POINTER :: SOL_loc
C     INTEGER, DIMENSION(:), POINTER :: ISOL_loc
C OUTPUT data and Statistics
C -----
C
C     INTEGER, DIMENSION(:), POINTER :: SYM_PERM, UNS_PERM
C     INTEGER INFO(40)
C     INTEGER INFOG(40) ! Global information (host only)
C
C     real RINFO(20)
C     real RINFOG(20) ! Global information (host only)
C   Schur
C     INTEGER SIZE_SCHUR, NPROW, NPCOL, MBLOCK, NBLOCK
C     INTEGER SCHUR_MLOC, SCHUR_NLOC, SCHUR_LLD
C     INTEGER, DIMENSION(:), POINTER :: LISTVAR_SCHUR
C
C     real/complex , DIMENSION(:), POINTER :: SCHUR
C   Mapping if provided by MUMPS
C     INTEGER, DIMENSION(:), POINTER :: MAPPING
C   Version number
C     CHARACTER(LEN=46) VERSION_NUMBER
C   Name of file to dump a problem in matrix market format
C     CHARACTER(LEN=255) WRITE_PROBLEM
C   Out-of-core
C     CHARACTER(LEN=63) :: OOC_PREFIX
C     CHARACTER(LEN=255) :: OOC_TMPDIR
C
C   END TYPE [SDCZ]MUMPS_STRUC

```

Figure 1: Main components of the ¹²structure [SDCZ]MUMPS_STRUC defined in [sdcz]mumps_struct.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`. Those components define the arguments to MUMPS that must be set by the user, or that are returned to the user.

4.1 Version number

`mumps_par%VERSION_NUMBER` (string) is set by MUMPS to the version number of MUMPS after a call to the initialization phase (`JOB=-1`).

For C users (see Section 8 for more general information), a macro `MUMPS_VERSION` is also defined in the include files `[sdcz]mumps_c.h`; it contains a string defining the version number. Typically, it is defined by: `#define MUMPS_VERSION "4.10.0"` This may be useful for users who wish to get the version number associated to the header file they include in their application (the component `VERSION_NUMBER` of the structure may be badly initialized in case of incompatible alignment options or incorrect version of the header file).

4.2 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 by MUMPS.

`JOB = -1` initializes an instance of the package. A call with `JOB = -1` must be performed before any other call to the package on the same instance. It sets default values for other components of `MUMPS_STRUC` (such as `ICNTL`, see below), 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`.

Note that if the user wants to modify one of those three components then he/she must destroy the instance (call with `JOB = -2`) then reinitialize the instance (call with `JOB = -1`).

Furthermore, after a call with `JOB = -1`, the internal component `mumps_par%MYID` contains the rank of the calling processor in the communicator provided to MUMPS. Thus, the test `“(mumps_par%MYID == 0)”` may be used to identify the host processor (see Section 2.6).

Finally, the version number is returned in `mumps_par%VERSION_NUMBER` (see Section 4.1).

`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 if `ICNTL(6)` is set to a value between 2 and 7. See `ICNTL(6)` in 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) ≠ 3`) (see Section 4.5),

- `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 4.6).

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 4.7 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 **A** provided by the user and the information from the analysis phase (`JOB=1`) to factorize the matrix **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) ≠ 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 4.5, 4.6, and 4.7.)

The actual pivot sequence used during the factorization may slightly differ from the sequence returned by the analysis if the matrix **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) ≠ 0`, see Section 4.8).

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 can also be used (see `ICNTL(25)`) to compute the null space basis of symmetric matrices provided that “null pivot row” detection (`ICNTL(24)`) was on and that the number of null pivots `INFOG(28)` was different from 0. It uses the right-hand side(s) **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 4.13.)

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.

4.3 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. Note that only the processors belonging to the communicator should call MUMPS.

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 and its value is communicated internally to the other phases as required. 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. Note that the value of **PAR** should be identical on all processors; if this is not the case, the value on processor 0 is used by the package.

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.

4.4 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. Its value is communicated internally to the other phases as required. Possible values for **SYM** are:

- 0 **A** is unsymmetric
- 1 **A** is suitable for symmetric positive definite since numerical pivoting is not performed and pivots are taken directly from the diagonal. In case ScaLAPACK is called, **P_POTRF** is used, which assumes positive diagonal pivots (an error -40 is returned in **INFOG(1)**). In case ScaLAPACK is not used (**ICNTL(13)**>0), this option will also work for more general classes of matrices, typically symmetric negative matrices. If the user thinks his matrix is positive definite, he/she may want to check that the number of negative pivots (**INFOG(12)**) is zero on exit. Another approach to suppress numerical pivoting which works with ScaLAPACK for both positive definite and negative definite matrices consists in setting **SYM=2** and **CNTL(1)=0.0D0** (recommended strategy).
- 2 **A** is general symmetric

Other values are treated as 0. Note that the value of **SYM** should be identical on all processors; if this is not the case, the value on processor 0 is used by the package. 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.

4.5 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**; they are not modified by the package.

- **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.

- 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) \neq 0. 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) or before analysis (JOB=1) if a numerical preprocessing option is requested (1 < ICNTL(6) < 7).

4.6 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. ELTPTR 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 $K_j + 1, \dots, K_j + L_j$, where
 - $K_j = \sum_{p=1}^{j-1} N_p^2$, and $L_j = N_j^2$ 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.

4.7 Distributed assembled matrix input: ICNTL(5)=0 and ICNTL(18) \neq 0

When the matrix is in assembled form (ICNTL(5)=0), we offer several options to distribute the matrix, 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 4.5.

mumps_par%N (integer), mumps_par%NZ (integer), mumps_par%NZ_loc (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 \mathbf{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 \mathbf{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 global¹ 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.
- NZ_loc is the dimension of the pointer array A_loc (see below).
- 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. Remark that $MAPPING$ is allocated by MUMPS, and not by the user. It will be freed during a call to MUMPS with $JOB = -2$.

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 $ICNTL(18)=1$.

4.8 Scaling: $ICNTL(8)$

$mumps_par\%COLSCA$, $mumps_par\%ROWSCA$ (double precision array pointers, dimension N) are optional, respectively column and row 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), in which case $ICNTL(8) = -2$ will be set by the package during analysis and should be passed unchanged to the solve phase ($JOB=3$).

4.9 Given ordering: $ICNTL(7)=1$

$mumps_par\%PERM_IN$ (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 $PERM_IN(i), i=1, \dots, N$ 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.

4.10 Schur complement with reduced (or condensed) right-hand side: $ICNTL(19)$ and $ICNTL(26)$

$mumps_par\%SIZE_SCHUR$ (integer) must be initialized by the user on the host to the number of variables defining the Schur complement if $ICNTL(19) = 1, 2, \text{ or } 3$. It is only accessed during the analysis phase and is not altered by MUMPS. Its value is communicated internally to the other phases as required. $SIZE_SCHUR$ should be greater or equal to 0 and strictly smaller than N .

¹If the calling application manages both local and global indices, the global indices must be provided.

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) = 1, 2 or 3**. It is not altered by MUMPS. It is accessed during analysis (**JOB=1**) and **LISTVAR_SCHUR(i)**, $i=1, \dots, \text{SIZE_SCHUR}$ must hold the i^{th} variable of the Schur complement matrix.

Centralized Schur complement stored by rows (**ICNTL(19)=1**)

Note that this option is becoming obsolete and is not recommended anymore because the memory for the Schur is doubled and because it requires a copy or message transfer of the Schur computed internally by MUMPS into the mumps_par%**SCHUR** argument. If a centralized Schur complement is required, we refer the user to the paragraph “Centralized Schur complement stored by columns (**ICNTL(19)=2 or 3**)” instead.

mumps_par%**SCHUR** is a **real (complex in the complex version)** 1-dimensional pointer array that should point to **SIZE_SCHUR × SIZE_SCHUR** locations in memory. It must be allocated by the user on the host (independently of the value of mumps_par%**PAR**) before the factorization phase. On exit, it holds the Schur complement matrix. 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. (This can also be viewed as the upper triangular part stored by columns in which case the lower part is not defined.)

Distributed Schur complement (**ICNTL(19)=2 or 3**)

For symmetric matrices, the value of **ICNTL(19)** controls whether only the lower part (**ICNTL(19) = 2**) or the complete matrix (**ICNTL(19) = 3**) is generated. MUMPS always provides the complete matrix for unsymmetric matrices so that either value for **ICNTL(19)** has the same effect.

If **ICNTL(19)=2 or 3**, the following parameters should be defined on the host on entry to the analysis phase :

mumps_par%**NPROW**, mumps_par%**NPCOL**, mumps_par%**MBLOCK**, and mumps_par%**NBLOCK** are integers corresponding to the characteristics of a 2D block cyclic grid of processors. They should be defined on the host before a call to the analysis phase. If any of these quantities is smaller than or equal to zero or has not been defined by the user, or if **NPROW × NPCOL** is larger than the number of slave processors available (total number of processors if **PAR=1**, total number of processors minus 1 if mumps_par%**PAR=0**), then a grid shape will be computed by the analysis phase of MUMPS and **NPROW**, **NPCOL**, **MBLOCK**, **NBLOCK** will be overwritten on exit from the analysis phase. Please refer to [13] (for example) for more details on the notion of grid of processors and on 2D block cyclic distributions. We briefly describe the meaning of the four above parameters here:

- **NPROW** is the number of rows of the process grid (or the number of processors in a column of the process grid),
- **NPCOL** is the number of columns of the process grid (or the number of processors in a row of the process grid),
- **MBLOCK** is the blocking factor used to distribute the rows of the Schur complement,
- **NBLOCK** is the blocking factor used to distribute the columns of the Schur complement.

As in ScaLAPACK, we use a row-major process grid of processors, that is, process ranks (as provided to MUMPS in the MPI communicator) are consecutive in a row of the process grid. **NPROW**, **NPCOL**, **MBLOCK** and **NBLOCK** should be passed unchanged from the analysis phase to the factorization phase. If the matrix is symmetric (**SYM=1 or 2**) and **ICNTL(19)=3** (see below), then the values of **MBLOCK** and **NBLOCK** should be equal.

On exit from the analysis phase, the following two components are set by MUMPS on the first **NPROW × NPCOL** slave processors (the host is excluded if **PAR=0** and the processors with largest MPI ranks in the communicator provided to MUMPS may not be part of the grid of processors).

mumps_par%**SCHUR_MLOC** is an integer giving the number of rows of the local Schur complement matrix on the concerned processor. It is equal to $\text{MAX}(1, \text{NUMROC}(\text{SIZE_SCHUR}, \text{MBLOCK}, \text{myrow}, 0, \text{NPROW}))$, where

- NUMROC is an INTEGER function defined in most ScaLAPACK implementations (also used internally by the MUMPS package),
- **SIZE_SCHUR**, **MBLOCK**, **NPROW** have been defined earlier, and
- *myrow* is defined as follows:
Let *myid* be the rank of the calling process in the communicator COMM provided to MUMPS. (*myid* can be returned by the MPI routine `MPI_COMM_RANK`.)
 - if **PAR** = 1 *myrow* is equal to $\text{myid} / \text{NPCOL}$,
 - if **PAR** = 0 *myrow* is equal to $(\text{myid} - 1) / \text{NPCOL}$.

Note that an upperbound of the minimum value of leading dimension (**SCHUR_LLD** defined below) is equal to $((\text{SIZE_SCHUR} + \text{MBLOCK} - 1) / \text{MBLOCK} + \text{NPROW} - 1) / \text{NPROW} * \text{MBLOCK}$.

mumps_par%**SCHUR_NLOC** is an integer giving the number of columns of the local Schur complement matrix on the concerned processor. It is equal to $\text{NUMROC}(\text{SIZE_SCHUR}, \text{NBLOCK}, \text{mycol}, 0, \text{NPCOL})$, where

- **SIZE_SCHUR**, **NBLOCK**, **NPCOL** have been defined earlier, and
- *mycol* is defined as follows:
Let *myid* be the rank of the calling process in the communicator COMM provided to MUMPS. (*myid* can be returned by the MPI routine `MPI_COMM_RANK`.)
 - if **PAR** = 1 *mycol* is equal to $\text{MOD}(\text{myid}, \text{NPCOL})$,
 - if **PAR** = 0 *mycol* is equal to $\text{MOD}(\text{myid} - 1, \text{NPCOL})$.

On entry to the factorization phase (**JOB** = 2), **SCHUR_LLD** should be defined by the user and **SCHUR** should be allocated by the user on the **NPROW** × **NPCOL** first slave processors (the host is excluded if **PAR**=0 and the processors with largest MPI ranks in the communicator provided to MUMPS may not be part of the grid of processors).

mumps_par%**SCHUR_LLD** is an integer defining the leading dimension of the local Schur complement matrix. It should be larger or equal to the local number of rows of that matrix, **SCHUR_MLOC** (as returned by MUMPS on exit from the analysis phase on the processors that participate in the computation of the Schur). **SCHUR_LLD** is not modified by MUMPS.

mumps_par%**SCHUR** is a **real** (**complex** in the complex version) one-dimensional pointer array that should be allocated by the user before a call to the factorization phase. Its size should be at least equal to $\text{SCHUR_LLD} \times (\text{SCHUR_NLOC} - 1) + \text{SCHUR_MLOC}$, where **SCHUR_MLOC**, **SCHUR_NLOC**, and **SCHUR_LLD** have been defined above. On exit to the factorization phase, the pointer array **SCHUR** contains the Schur complement, stored by columns, in the format corresponding to the 2D cyclic grid of **NPROW** × **NPCOL** processors, with block sizes **MBLOCK** and **NBLOCK**, and local leading dimensions **SCHUR_LLD**.

The Schur complement is stored by columns. Note that setting $\text{NPCOL} \times \text{NPROW} = 1$ will centralize the Schur complement matrix, *stored by columns* (instead of by rows as in the **ICNTL(19)=1** option). It will then be available on the host node if **PAR**=1, and on the node with MPI identifier 1 (first working slave processor) if **PAR**=0. More details on this are presented in the paragraph below.

If **ICNTL(19)=2** and the Schur is symmetric (**SYM**=1 or 2), only the lower triangle is provided, stored by columns.

If **ICNTL(19)=3** and the Schur is symmetric (**SYM**=1 or 2), then both the lower and upper triangles are provided, stored by columns. Note that if **ICNTL(19)=3**, then the constraint $\text{mumps_par}\% \text{MBLOCK} = \text{mumps_par}\% \text{NBLOCK}$ should hold.

(For unsymmetric matrices, **ICNTL(19)=2** and **ICNTL(19)=3** have the same effect.)

Centralized Schur complement stored by columns (**ICNTL(19)=2 or 3**)

This option is recommended compared to **ICNTL(19)=1**. It is a particular case of the distributed Schur complement (**ICNTL(19)=2 or 3**, see the above paragraph), where the Schur complement is only assigned to one processor. Therefore we refer the reader to the previous section for a detailed description of using this option. Let us summarize it a simple case of use, where the user wants a centralized Schur complement and where **PAR=1** (working host node).

On top of **SIZE_SCHUR** and **LISTVAR_SCHUR** described earlier, the user should set the following parameters on the host on entry to the analysis phase:

NPROW = NPCOL = 1, in order to define a distribution that uses only one processor (the host assuming that **PAR=1**);

MBLOCK = NBLOCK = 100. Those arguments must be provided and be strictly positive but their actual value will not change the distribution since **NPROW=NPCOL=1**.

ICNTL(19)=2 or 3.

On entry to the factorization phase, the user should provide on the host²:

mumps_par%SCHUR_LLD=SIZE_SCHUR: we consider here the simple case where the leading dimension of the Schur is equal to its order.

mumps_par%SCHUR, a **real (complex in the complex version)** one-dimensional pointer array of size **SIZE_SCHUR × SIZE_SCHUR** that should be allocated by the user.

On exit to the factorization phase, the pointer array **SCHUR** available on the host contains the Schur complement. If **SYM=0**, then the options of **ICNTL(19)=2** and **ICNTL(19)=3** have an identical behaviour and the asymmetric Schur complement is returned by columns (i.e., in column-major format). If **SYM=1 or 2** and **ICNTL(19)=2**, then only the lower triangular part of the symmetric Schur is returned, stored by columns, and the upper triangular part should not be accessed. (Note that this is equivalent to say that the upper triangular part is returned by rows and the lower triangular part is not accessed.) If **SYM=1 or 2** and **ICNTL(19)=3**, then both the lower and upper triangular parts are returned. Because the Schur complement is symmetric, this can be seen both as a row-major and as a column-major storage.

Using partial factorization during solution phase (**ICNTL(26)= 0, 1 or 2**)

As explained in Section 2.12, when a Schur complement has been computed during the factorization phase, then either the solution phase computes a solution on the internal problem (**ICNTL(26)=0**, see control parameter **ICNTL(26)**), or the complete problem can use a reduced right-hand side to build the solution of the problem on the Schur variables (**ICNTL(26)=1** and **ICNTL(26)=2**).

If **ICNTL(26)=1 or 2**, then the following parameters must be defined on the host on entry to the solution step:

mumps_par%LREDRHS is an optional integer parameter defining the leading dimension of the reduced right-hand side, **REDRHS**, that must be set by the user when **NRHS** is provided. It must be larger or equal to **SIZE_SCHUR**, the size of the Schur complement.

mumps_par%REDRHS is a **real (complex in the complex version)** one-dimensional pointer array that should be allocated by the user before entering the solution phase. Its size should be at least equal to **LREDRHS × (NRHS-1) + SIZE_SCHUR**. If **ICNTL(26)=1**, then on exit from the solution phase, **REDRHS(i+(k-1)*LREDRHS)**, $i=1, \dots, \text{SIZE_SCHUR}$, $k=1, \dots, \text{NRHS}$ will hold the reduced right-hand side. If **ICNTL(26)=2**, then **REDRHS(i+(k-1)*LREDRHS)**, $i=1, \dots, \text{SIZE_SCHUR}$, $k=1, \dots, \text{NRHS}$ must be set (on entry to the solution phase) to the solution on the Schur variables. In that case (ie, **ICNTL(26)=2**), it is not altered by **MUMPS**.

²As said above, we assume a working host model (**PAR=1**), otherwise this becomes processor 1 – please refer to the general description from paragraph “Distributed Schur Complement” above for more information.

4.11 Out-of-core ($ICNTL(22) \neq 0$)

The decision to use the disk to store the matrix of factors is controlled by $ICNTL(22)$ ($ICNTL(22) \neq 0$ implies out-of-core). Only the value on the host node is significant.

Both $mumps_par\%OOC_TMPDIR$ and $mumps_par\%OOC_PREFIX$ can be provided by the user (on each processor) to control respectively the directory where the out-of-core files will be stored and the prefix of those files. If not provided, the `/tmp` directory will be tried and file names will be chosen automatically.

It is also possible to provide the directory and filename prefix through environment variables. If $mumps_par\%OOC_TMPDIR$ is not defined, then MUMPS checks for the environment variable `MUMPS_OOC_TMPDIR`. If not defined, then the directory `/tmp` is attempted. Similarly, if $mumps_par\%OOC_PREFIX$ is not defined, then MUMPS checks for the environment variable `MUMPS_OOC_PREFIX`. If not defined, then MUMPS chooses the filename automatically.

4.12 Workspace parameters

The memory required to run the numerical phases is estimated during the analysis. The size of the workspace required during numerical factorization depends on algorithmic parameters such as the in-core/out-of-core strategies ($ICNTL(22)$) and the memory relaxation parameter $ICNTL(14)$.

Two main integer and real/complex workarrays (IS and S, respectively) that hold factors, active frontal matrices, and contribution blocks are allocated internally. Note that, apart from these two large work arrays, other internal work arrays exist (for example, internal communication buffers in the parallel case, or integer arrays holding the structure of the assembly tree).

At the end of the analysis phase, the following estimations of the memory required to run the numerical phases are provided (for the given or default value of the memory relaxation parameter $ICNTL(14)$):

- $INFO(15)$ returns the minimum size in Megabytes to run the numerical phases (factorisation/solve) $\boxed{in-core}$. (The maximum and sum over all processors are returned respectively in $INFOG(16)$ and $INFOG(17)$).
- $INFO(17)$ provides an estimation (in Megabytes) of the minimum total memory required to run the numerical phases $\boxed{out-of-core}$. (The maximum and sum over all processors are returned respectively in $INFOG(26)$ and $INFOG(27)$).

Those memory estimations can be used as lower bounds when the user wants to explicitly control the memory used (see description of $ICNTL(23)$).

As a first general approach, we advise the user to rely on the estimations provided during the analysis phase. If the user wants to increase the allocated workspace (typically, numerical pivoting that leads to extra storage, or previous call to MUMPS that failed because of a lack of allocated memory), we describe in the following how the size of the workspace can be controlled.

- The memory relaxation parameter $ICNTL(14)$ is designed to control the increase, with respect to the estimations performed during analysis, in the size of the workspace allocated during the numerical phase.
- The user can also provide the size of the total memory $ICNTL(23)$ that the package is allowed to use internally. $ICNTL(23)$ is expressed in Megabytes per processor. If $ICNTL(23)$ is provided, $ICNTL(14)$ is still used to relax the integer workspace and some internal buffers. That memory is subtracted from $ICNTL(23)$; what remains determines the size of the main (and most memory-consuming) real/complex array holding the factors and stack of contribution blocks.

4.13 Right-hand side and solution vectors/matrices

The formats of the right-hand side and of the solution are controlled by $ICNTL(20)$ and $ICNTL(21)$, respectively.

Centralized dense right-hand side (**ICNTL(20)=0**) and/or centralized dense solution (**ICNTL(21)=0**)

If **ICNTL(20)=0** or **ICNTL(21)=0**, the following components of the MUMPS structure should be defined on the host.

mumps_par%**RHS** (**real/complex** array pointer, dimension **LRHS**×**NRHS**) is a **real (complex** in the complex version) array that should be allocated by the user on the host before a call to MUMPS with **JOB= 3, 5, or 6**.

On entry, if **ICNTL(20)=0**, **RHS(i+(k-1)×LRHS)** must hold the *i*-th component of *k*th right-hand side vector ($1 \leq k \leq \text{NRHS}$) of the equations being solved. The default value for **NRHS** is 1 and the default value for **LRHS** is **N**, the order of the matrix (see below).

On exit, if **ICNTL(21)=0**, then **RHS(i+(k-1)×LRHS)** will hold the *i*-th component of the *k*th solution vector, $1 \leq k \leq \text{NRHS}$. Remark that when **ICNTL(20)=0** (dense right-hand side) and **ICNTL(21)=1** (distributed solution), **RHS** may still be modified on exit to the package, but will not contain any significant data for the user.

mumps_par%**NRHS** (integer) is an optional parameter that is significant on the host before a call to the solution phase of MUMPS (**JOB = 3, 5, or 6**). If **NRHS** is set, it should hold the number of right-hand side vectors. If not set, the value 1 is assumed, as this ensures backward compatibility of the MUMPS interface with versions of the package prior to 4.3.3. Note that if **NRHS > 1**, then functionalities related to iterative refinement and error analysis (see **ICNTL(10)** and **ICNTL(11)**) are currently disabled.

mumps_par%**LRHS** (integer) is an optional parameter that is significant on the host before a call to the solve phase of MUMPS (**JOB=3, 5, or 6**). If **NRHS** is provided, **LRHS** should then hold the leading dimension of the array **RHS** and should be greater than or equal to **N**.

Sparse right-hand side (**ICNTL(20)=1** or **ICNTL(30)=1**)

If **ICNTL(20)=1** or **ICNTL(30)=1**, the following input parameters should be defined on the host only before a call to MUMPS with **JOB=3, 5, or 6**:

mumps_par%**NZ_RHS** (integer) should hold the total number of non-zeros in all the right-hand side vectors.

mumps_par%**NRHS** (integer), if set, should hold the number of right-hand side vectors. If not set, the value 1 is assumed.

mumps_par%**RHS_SPARSE** (**real/complex** array pointer, dimension **NZ_RHS**) should hold the numerical values of the non-zero inputs of each right-hand side vector when **ICNTL(20)=1**. It must be allocated but needs not be initialized when **ICNTL(30)=1**. See also **IRHS_PTR** below.

mumps_par%**IRHS_SPARSE** (integer array pointer, dimension **NZ_RHS**) should hold the indices of the variables of the non-zero inputs of each right-hand side vector.

mumps_par%**IRHS_PTR** is an integer array pointer of dimension **NRHS+1**. **IRHS_PTR** is such that the *i*-th right-hand side vector is defined by its non-zero row indices **IRHS_SPARSE(IRHS_PTR(i)...IRHS_PTR(i+1)-1)** and the corresponding numerical values **RHS_SPARSE(IRHS_PTR(i)...IRHS_PTR(i+1)-1)**. Note that **IRHS_PTR(1)=1** and **IRHS_PTR(NRHS+1)=NZ_RHS+1**.

Note that, if the right-hand side is sparse and if the solution is centralized (**ICNTL(21)=0**) and if the computation of entries in \mathbf{A}^{-1} is not requested (**ICNTL(30)=0**), then mumps_par%**RHS** should still be allocated on the host, as explained in the previous section. On exit from a call to MUMPS with **JOB=3, 5, or 6**, it will hold the centralized solution. More explicitly, with **ICNTL(30)=1** mumps_par%**RHS** needs not be allocated and since selected entries of the inverse of the matrix are requested, mumps_par%**NRHS** must be set to **N** and column *j* of the sparse right-hand side, that might be empty, corresponds to column *j* of \mathbf{A}^{-1} .

Distributed solution (ICNTL(21)=1)

On some networks with low bandwidth, and especially when there are many right-hand side vectors, centralizing the solution on the host processor might be a costly part of the solution phase. If this is critical to the user, this functionality allows the solution to be left distributed over the processors. The solution should then be exploited in its distributed form by the user application.

mumps_par%**SOL_loc** is a **real/complex** array pointer, of dimension **LSOL_loc**× **NRHS** (where **NRHS** corresponds to the value provided in mumps_par%**NRHS** on the host), that should be allocated by the user before the solve phase (**JOB**=3) 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). Its leading dimension **LSOL_loc** should be larger than or equal to **INFO(23)**, where **INFO(23)** has the value returned by MUMPS on exit from the factorization phase. On exit from the solve phase, **SOL_loc(i+(k-1)×LSOL_loc)** will contain the value corresponding to variable **ISOL_loc(i)** in the k^{th} solution vector.

mumps_par%**LSOL_loc** (integer). **LSOL_loc** must be set to the leading dimension of **SOL_loc** (see above) and should be larger than or equal to **INFO(23)**, where **INFO(23)** has the value returned by MUMPS on exit from the factorization phase.

mumps_par%**ISOL_loc** (integer array pointer, dimension **INFO(23)**) **ISOL_loc** should be allocated by the user before the solve phase (**JOB**=3) 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). **ISOL_loc** should be of size at least **INFO(23)**, where **INFO(23)** has the value returned by MUMPS on exit from the factorization phase. On exit from the solve phase, **ISOL_loc(i)** contains the index of the variables for which the solution (in **SOL_loc**) is available on the local processor. Note that if successive calls to the solve phase (**JOB**=3) are performed for a given matrix, **ISOL_loc** will have the same contents for each of these calls.

Note that if the solution is kept distributed, then functionalities related to error analysis and iterative refinement (see **ICNTL(10)** and **ICNTL(11)**) are currently not available.

4.14 Writing a matrix to a file

mumps_par%**WRITE_PROBLEM** (string) can be set by the user before the analysis phase (**JOB**=1) in order to write the matrix passed to MUMPS into the file “WRITE.PROBLEM”. This only applies to assembled matrices and the format used to write the matrix is the “matrix market” format³. If the matrix is distributed, then each processor must initialize **WRITE_PROBLEM**. Each processor will then write its share of the matrix in a file whose name is “WRITE.PROBLEM” appended by the rank of the processor in the communicator passed to MUMPS. Note that **WRITE_PROBLEM** should include both the path and the file name.

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.

5.1 Integer control parameters

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.

³See <http://math.nist.gov/MatrixMarket/>

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, warnings, and main statistics printed.
- 3 : Errors and warnings and terse diagnostics (only first ten entries of arrays) printed.
- ≥ 4 : Errors and warnings and 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 4.7). If ICNTL(5) = 1, the input matrix must be given in elemental format in the structure components N, NELT, ELTPTR, ELTVAR, and A_ELT. Values of ICNTL(5) different from 0 and 1 are treated as 0.

Please note that parallel analysis is only available for matrices in assembled format and, thus, an error will be raised if ICNTL(5)=1 and ICNTL(28)=2.

ICNTL(6) has default value 7 (automatic choice done by the package) and is used to control an option for permuting and/or scaling the matrix. It is only accessed by the host and only during the analysis phase. For unsymmetric matrices, if ICNTL(6)=1, 2, 3, 4, 5, 6 a column permutation (based on weighted bipartite matching algorithms described in [17, 18]) is applied to the original matrix to get a zero-free diagonal. For symmetric matrices, if ICNTL(6)=1, 2, 3, 4, 5, 6, the column permutation is not applied but it can be used to determine a set of recommended 1×1 and 2×2 pivots (see [19] for more details).

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 permutation is such that 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 (if permutation was applied) is maximized.
- 5 : The product of the diagonal entries of the permuted matrix (if permutation was applied) is maximized. Vectors are computed (and stored in COLSCA and ROWSCA, only if ICNTL(8) is set to -2 or 77) to scale the matrix. In case the matrix is effectively permuted (unsymmetric matrix) then the nonzero diagonal entries in the permuted matrix are one in absolute value and all the off-diagonal entries 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, 1 or 7, the numerical values of the original matrix, `mumps_par%A`, must be provided by the user during the analysis phase. If the matrix is symmetric positive definite (`SYM = 1`), or in elemental format (`ICNTL(5)=1`), or the ordering is provided by the user (`ICNTL(7)=1`), or the Schur option (`ICNTL(19) = 1, 2, or 3`) is required, or the matrix is initially distributed (`ICNTL(18) \neq 0`), then ICNTL(6) is treated as 0.

On unsymmetric matrices (`SYM = 0`), the user is advised to set ICNTL(6) to a nonzero value when the matrix is very unsymmetric in structure. 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 on the host processor.

(The column permutation is such that entry $a_{i,perm(i)}$ is on the diagonal of the permuted matrix.) Otherwise, the pointer is unassociated.

On general symmetric matrices ($SYM = 2$), we advise either to let MUMPS select the strategy ($ICNTL(6) = 7$) or to set $ICNTL(6) = 5$ if the user knows that the matrix is for example an augmented system (which is a system with a large zero diagonal block). On output from the analysis the pointer `mumps_par%UNS_PERM` is unassociated.

On output from the analysis phase, `INFOG(23)` holds the value of $ICNTL(6)$ that was effectively used.

Please note that this permutation/scaling of the matrix is incompatible with parallel analysis and, thus an error will be raised if $ICNTL(28)=2$ and $ICNTL(6)=1,2,3,4,5$, or 6.

$ICNTL(7)$ has default value 7 and is only accessed by the host and only during the analysis phase. If sequential analysis is to be performed ($ICNTL(28)=1$), 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. In exceptional cases, The option corresponding to $ICNTL(7)$ may be forced by MUMPS when the ordering suggested by the user is not compatible with the value of $ICNTL(12)$. Possible values for $ICNTL(7)$ are:

- 0 : Approximate Minimum Degree (AMD) [4] is used,
- 1 : the pivot order should be set by the user in `PERM_IN`, on the host processor. In that case, `PERM_IN` must be allocated on the host by the user and `PERM_IN(i)`, ($i=1, \dots, N$) must hold the position of variable i in the pivot order.
- 2 : Approximate Minimum Fill (AMF) is used,
- 3 : SCOTCH⁴ [28] is used (if previously installed by the user).
- 4 : PORD⁵ [31] is used,
- 5 : the METIS⁶ [26] package is used (if previously installed by the user),
- 6 : Approximate Minimum Degree with automatic quasi-dense row detection (QAMD) is used.
- 7 : Automatic choice 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 then options 3, 4 and 5 are treated as 7.

- If the user asks for a Schur complement matrix and the matrix is assembled then only options 0, 1, 5 and 7 are currently available. Other options are treated as 7.
- For `elemental matrices` ($ICNTL(5)=1$), only options 0, 1, 5 and 7 are available, with option 7 leading to an automatic choice between AMD and METIS (options 0 or 5); other values are treated as 7. Furthermore, if the user asks for a Schur complement matrix, only options 0, 1 and 7 are currently available. Other options are treated as 7 which will (currently) be treated as 0 (AMD).

Generally, with the automatic choice corresponding to $ICNTL(7)=7$, the option 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 matrices with relatively dense rows, we highly recommend option 6 which may significantly reduce the time for analysis.

On output, the pointer `mumps_par%SYM_PERM` provides access, on the host processor, to the symmetric permutation that is effectively used by the MUMPS package, and `INFOG(7)` to the ordering option that was effectively used. (`mumps_par%SYM_PERM(i)`, ($i=1, \dots, N$) holds the position of variable i in the pivot order.)

Please note that $ICNTL(7)$ is meaningless if the parallel analysis is chosen, i.e., $ICNTL(28)=2$.

⁴See <http://gforge.inria.fr/projects/scotch/> to obtain a copy.

⁵Distributed within MUMPS by permission of J. Schulze (University of Paderborn).

⁶See <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview> to obtain a copy.

ICNTL(8) has default value 77.

It is used to describe the scaling strategy and is only accessed by the host.

If $\text{ICNTL}(8) = -1$, scaling vectors must be provided in COLSCA and ROWSCA by the user, who is then responsible for allocating and freeing them, If $\text{ICNTL}(8) = 0$, no scaling is performed, and arrays COLSCA/ROWSCA are not used. Otherwise, the scaling arrays COLSCA/ROWSCA are allocated and computed by the package.

If $\text{ICNTL}(8) = 77$, then an automatic choice of the scaling option may be performed, either during the analysis or the factorization. The effective value used for $\text{ICNTL}(8)$ is returned in INFOG(33). If the scaling arrays are computed during the analysis, then they are ready to be used by the factorization stage. Note that scalings can be efficiently computed during analysis when requested (see [ICNTL\(6\)](#) and [ICNTL\(12\)](#)).

Possible values of $\text{ICNTL}(8)$ are listed below:

- -2: Scaling computed during analysis (see [\[17, 18\]](#) for the unsymmetric case and [\[19\]](#) for the symmetric case).
- -1: Scaling arrays provided on entry to the numerical factorization phase.
- 0 : No scaling applied/computed.
- 1 : Diagonal scaling computed during the numerical factorization phase,
- 2 : Row and column scaling based on [\[14\]](#), computed during the numerical factorization phase,
- 3 : Column scaling computed during the numerical factorization phase,
- 4 : Row and column scaling based on infinite row/column norms, computed during the numerical factorization phase,
- 5 : Scaling based on [\[14\]](#) followed by column scaling; computed during the numerical factorization phase,
- 6 : Scaling based on [\[14\]](#) followed by row and column scaling; computed during the numerical factorization phase.
- 7 : Simultaneous row and column iterative scaling based on [\[30\]](#) and [\[10\]](#); computed during the numerical factorization phase.
- 8 : Similar to 7 but more rigorous and expensive to compute; computed during the numerical factorization phase.
- 77 (analysis only) : Automatic choice of $\text{ICNTL}(8)$ value done during analysis.

If the input matrix is symmetric ($\text{SYM} \neq 0$), then only options -2, -1, 0, 1, 7, 8 and 77 are allowed and other options are treated as 0; if $\text{ICNTL}(8) = -1$, the user should ensure that the array ROWSCA is equal to (or points to the same location as) the array COLSCA. If the input matrix is in elemental format ($\text{ICNTL}(5) = 1$), then only options -1 and 0 are allowed and other options are treated as 0. If the initial matrix is distributed ($\text{ICNTL}(18) \neq 0$ and $\text{ICNTL}(5) = 0$), then only options 7, 8 and 77 are allowed, otherwise no scaling is applied. If $\text{ICNTL}(8) = -2$ then the user has to provide the numerical values of the original matrix (mumps_par%A) on entry to the analysis.

ICNTL(9) has default value 1 and is only accessed by the host during the solve phase. If $\text{ICNTL}(9) = 1$, $\mathbf{Ax} = \mathbf{b}$ 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. If $\text{NRHS} = 1$, then $\text{ICNTL}(10)$ corresponds to the maximum number of steps of iterative refinement. If $\text{ICNTL}(10) \leq 0$, iterative refinement is not performed.

In the current version, if $\text{ICNTL}(21)=1$ (solution kept distributed), or if $\text{NRHS} > 1$, then iterative refinement is not performed and $\text{ICNTL}(10)$ is treated as 0.

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 $\text{ICNTL}(9)$): the infinite norm of the input matrix, the computed solution, and the scaled residual in RINFOG(4), RINFOG(5) and 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 linear system in RINFOG(10) and RINFOG(11). See also Section 2.3. Note that if performance is critical, $\text{ICNTL}(11)$ should be kept equal to 0. Finally,

note that, if $NRHS > 1$, or if $ICNTL(21)=1$ (solution vector kept distributed) then error analysis is not performed and $ICNTL(11)$ is treated as 0.

$ICNTL(12)$ is meaningful only on general symmetric matrices ($SYM = 2$) and its default value is 0 (automatic choice). For unsymmetric matrices ($SYM=0$) or symmetric definite positive matrices ($SYM=1$) all values of $ICNTL(12)$ are treated as 1 (nothing done). It is only accessed by the host and only during the analysis phase. It defines the ordering strategy (see [19] for more details) and is used, in conjunction with $ICNTL(6)$, to add constraints to the ordering algorithm ($ICNTL(7)$ option). Possible values of $ICNTL(12)$ are :

- 0 : automatic choice
- 1 : usual ordering (nothing done)
- 2 : ordering on the compressed graph associated with the matrix.
- 3 : constrained ordering, only available with AMF ($ICNTL(7)=2$).

Other values are treated as 0. $ICNTL(12)$, $ICNTL(6)$, $ICNTL(7)$ values are strongly related. As for $ICNTL(6)$, if the matrix is 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 $ICNTL(12)$ is treated as one.

If MUMPS detects some incompatibility between control parameters then it uses the following rules to automatically reset the control parameters. Firstly $ICNTL(12)$ has a lower priority than $ICNTL(7)$ so that if $ICNTL(12) = 3$ and the ordering required is not AMF then $ICNTL(12)$ is internally treated as 2. Secondly $ICNTL(12)$ has a higher priority than $ICNTL(6)$ and $ICNTL(8)$. Thus if $ICNTL(12) = 2$ and $ICNTL(6)$ was not active ($ICNTL(6)=0$) then $ICNTL(6)$ is treated as 5 if numerical values are provided, or as 1 otherwise. Furthermore, if $ICNTL(12) = 3$ then $ICNTL(6)$ is treated as 5 and $ICNTL(8)$ is treated as -2.

On output from the analysis phase, $INFOG(24)$ holds the value of $ICNTL(12)$ that was effectively used. Note that $INFOG(7)$ and $INFOG(23)$ hold the values of $ICNTL(7)$ and $ICNTL(6)$ (respectively) that were effectively used.

$ICNTL(13)$ has default value 0 and is only accessed by the host during the analysis phase. If $ICNTL(13) \leq 0$, ScaLAPACK will be used for the root frontal matrix (last Schur complement to be factored) if its size is larger than a machine-dependent minimum size. Otherwise ($ICNTL(13) > 0$), ScaLAPACK will not be used and the root node will be treated sequentially. Processing the root sequentially can be useful when the user is interested in the inertia of the matrix (see $INFO(12)$ and $INFOG(12)$), or when the user wants to detect null pivots (see $ICNTL(24)$).

This parameter also controls splitting of the root frontal matrix. If the number of working processors is strictly larger than $ICNTL(13)$ with $ICNTL(13) > 0$ (ScaLAPACK off), then splitting of the root node is performed, in order to automatically recover part of the parallelism lost because the root node was processed sequentially. Finally, setting $ICNTL(13)$ to -1 will force splitting of the root node in all cases (even sequentially), while values strictly smaller than -1 will be treated as 0.

Note that, although $ICNTL(13)$ controls the efficiency of the factorization and solve phases, preprocessing work is performed during analysis and this option must be set on entry to the analysis phase.

$ICNTL(14)$ 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, increasing $ICNTL(14)$ may help. Except in special cases, the default value is 20 (which corresponds to a 20 % increase).

$ICNTL(15-17)$ Not used in current version.

$ICNTL(18)$ 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 4.5.
- 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.

Other values are treated as 0. For options 1, 2, 3, see Section 4.7 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)=1, then the Schur complement matrix will be returned to the user on the host after the factorization phase. If ICNTL(19)=2 or 3, then the Schur will be returned to the user on the slave processors in the form of a 2D block cyclic distributed matrix (ScaLAPACK style). Values not equal to 1, 2 or 3 are treated as 0. If ICNTL(19) equals 1, 2, or 3, the user must set on entry to the analysis phase, on the host node:

- 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.

For a distributed Schur complement (ICNTL(19)=2 or 3), the integer variables `NPROW`, `NPCOL`, `MBLOCK`, `NBLOCK` may also be defined on the host before the analysis phase (default values will otherwise be provided). Furthermore, workspace should be allocated by the user before the factorization phase in order for MUMPS to store the Schur complement (see `SCHUR`, `SCHUR_MLOC`, `SCHUR_NLOC`, and `SCHUR_LLD` in Section 4.10).

Note that the partial factorization of the interior variables can then be exploited to perform a solve phase (transposed matrix or not, see ICNTL(9)). Note that the right-hand side (`RHS`) provided on input must still be of size `N` (or $N \times \text{NRHS}$ in case of multiple right-hand sides) 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, since the Schur complement is 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 and parallel analysis. If the ordering is given (ICNTL(7)=1) then the following property should hold: `PERM_IN(LISTVAR_SCHUR(i)) = N-mumpsSIZE_SCHUR+i`, for $i=1, \text{SIZE_SCHUR}$.

ICNTL(20) has default value 0 and is only accessed by the host during the solve phase. If ICNTL(20)=0, the right-hand side must be given in dense form in the structure component `RHS`. If ICNTL(20)=1, then the right-hand side must be given in sparse form using the structure components `IRHS_SPARSE`, `RHS_SPARSE`, `IRHS_PTR` and `NZ_RHS`. Values of ICNTL(20) that are different from 0 and 1 are treated as 0. (See Section 4.13).

ICNTL(21) has default value 0 and is only accessed by the host during the solve phase. If ICNTL(21)=0, the solution vector will be assembled and stored in the structure component `RHS`, that must have been allocated earlier by the user. If ICNTL(21)=1, the solution vector is kept distributed at the end of the solve phase, and will be available on each slave processor in the structure components `ISOL_loc` and `SOL_loc`. `ISOL_loc` and `SOL_loc` must then have been allocated by the user and must be of size at least `INFO(23)`, where `INFO(23)` has been returned by MUMPS at the end of the factorization phase. Values of ICNTL(21) different from 0 and 1 are currently treated as 0.

Note that if the solution is kept distributed, error analysis and iterative refinement (controlled by `ICNTL(10)` and `ICNTL(11)`) are not applied.

ICNTL(22) has default value 0 and controls the in-core/ out-of-core (OOC) facility. It must be set on the host before the factorization phase. Possible values are:

- 0: In core factorization and solution phases (default standard version).
- 1: Out of core factorization and solve phases. The complete matrix of factors is written to disk (see Section 4.11).

ICNTL(23) has default value 0. It can be provided by the user at the beginning of the factorization phase and is only significant on the host. It corresponds to the maximum size of the working memory in MegaBytes that MUMPS can allocate per working processor. (It covers all internal integer and real (complex in the complex version) workspace.)

If ICNTL(23) is greater than 0 then MUMPS automatically computes the size of the internal workarrays such that the storage for all MUMPS internal data is equal to ICNTL(23). The relaxation ICNTL(14) is first applied to the internal integer workarray IS and to communication and I/O buffers; the remaining available space is given to the main (and most critical) real/complex internal workarray S holding the factors and the stack of contribution blocks. A lower bound of ICNTL(23) (if ICNTL(14) has not been modified since the analysis) is given by INFOG(16) if the factorization is in-core (ICNTL(22)=0), and by INFOG(26) if the factorization is out-of-core (ICNTL(22)=1).

If ICNTL(23) is left to its default value 0 then each processor will allocate workspace based on the estimates computed during the analysis (INFO(17) if ICNTL(14) has not been modified since analysis, or larger if ICNTL(14) was increased). Note that these estimates are accurate in the sequential version of MUMPS, but that they can be inaccurate in the parallel case, especially for the out-of-core version. Therefore, in parallel, we recommend to use ICNTL(23) and provide a value significantly larger than INFOG(16) in the in-core case, or INFOG(26) in the out-of-core case.

ICNTL(24) has default value 0 and controls the detection of “null pivot rows”. Null pivot rows are modified to enable the solution phase to provide one solution among the possible solutions of the numerically deficient matrix. Note that the list of row indices corresponding to null pivots is returned on the host in PIVNUL_LIST(1:INFOG(28)). The solution phase (JOB=3) can then be used to either provide a “regular” solution (in the sense that it is a possible solution of the complete system when the right-hand-side belongs to the span of the original matrix) or to compute the associated vectors of the null-space basis (see ICNTL(25), case of symmetric matrices only). Possible values of ICNTL(24) are:

- 0 Nothing done. A null pivot will result in error INFO(1)=-10.
- 1 Null pivot row detection; CNTL(3) is used to compute the threshold to decide that a pivot row is “null”. The parameter CNTL(5) then defines the fixation that will be used to enable the solution phase to provide a possible solution to the original system.

Other values are treated as 0. Note that when ScaLAPACK is applied on the root node (see ICNTL(13)), then exact null pivots on the root will stop the factorization (INFO(1)=-10) while tiny pivots on the root node will still be factored. Setting ICNTL(13) to a non-zero value will help with the correct detection of null pivots but degrade performance.

ICNTL(25) has default value 0 and is only accessed by the host during the solution phase. It allows the computation of a null space basis, which is meaningful only if a zero-pivot detection option was requested (ICNTL(24) \neq 0) during the factorization and if the matrix was found to be deficient (INFOG(28) > 0); This functionality is only available for symmetric matrices and is not currently available in the case of unsymmetric matrices. Possible values of ICNTL(25) are:

- 0 A normal solution step is performed. If the matrix was found singular during factorization then one possible solution is returned.
- i with $1 \leq i \leq \text{INFOG}(28)$. The i -th vector of the null space basis is computed.
- -1. The complete null space basis is computed.
- Other values result in an error.

Note that when vectors from the null space are requested, both centralized and distributed solutions options can be used. In both cases space, to store the null space, vectors must be allocated by the user and provided to MUMPS. If the solution is centralized (ICNTL(21)=0), then the null space vectors are returned to the user in the array RHS, allocated by the user on the host. If the solution is distributed (ICNTL(21)=1), then the null space vectors are returned in the array SOL_loc. In both cases, note that the number of columns of RHS or SOL_loc must be equal to the number of vectors requested, so that NRHS is equal to:

- 1 if $1 \leq \text{ICNTL}(25) \leq \text{INFOG}(28)$;
- INFOG(28) if ICNTL(25)=-1.

Finally, note that iterative refinement, error analysis, and the option to solve the transpose system (ICNTL(9)) are ignored when the solution step is used to return vectors from the null space (ICNTL(25) \neq 0).

ICNTL(26) has default value 0 and is accessed by the host during the solution phase. ICNTL(26) is only meaningful if combined with the Schur option (ICNTL(19) \neq 0, see above). It can be used to condense/reduce (ICNTL(26)=1) the right-hand side on the Schur variables, or to expand (ICNTL(26)=2) the Schur local solution on the complete solution (see Section 2.12).

If ICNTL(26) \neq 0, then the user should provide workspace in the pointer array REDRHS, as well as a leading dimension LREDRHS (see Section 4.10).

If ICNTL(26)=1 then only a forward elimination is performed. The solution corresponding to the "internal" (non-Schur) variables is returned together with the reduced/condensed right-hand-side. The reduced right-hand side is made available on the host in REDRHS.

If ICNTL(26)=2 then REDRHS is considered to be the solution corresponding to the Schur variables. The backward substitution is then performed with the given right-hand side to compute the solution associated with the "internal" variables. Note that the solution corresponding to the Schur variables is also made available in the main solution vector/matrix.

Values different from 1 and 2 are treated as 0. Note that if no Schur complement was computed, ICNTL(26) = 1 or 2 results in an error. Finally, if ICNTL(26) = 1 or 2, then error analysis and iterative refinements are disabled.

ICNTL(27) *Experimental parameter subject to change in a future release.* ICNTL(27) is only accessed by the host during the solution phase. It controls the blocking size for multiple right-hand sides. It influences both the memory usage (see INFOG(30) and INFOG(31)) and the solution time. Larger values of ICNTL(27) lead to larger memory requirements and a better performance (except if the larger memory requirements induce swapping effects). Tuning ICNTL(27) is critical, especially when factors are on disk (ICNTL(22)=1 at the factorization stage) because factors must be accessed once for each block of right-hand sides. A negative value indicates that an automatic setting is performed by the solver: when ICNTL(27) is negative, the blocksize is currently set to (i) $-2 \times \text{ICNTL}(27)$ if the factors are on disk (ICNTL(22)=1); and to (ii) $-\text{ICNTL}(27)$ otherwise (in-core factors). The default value is -8 and zero is treated as one.

ICNTL(28) This parameter is only accessed by the host process during the analysis phase and decides whether a parallel or a sequential analysis will be performed. Three values are possible:

- 0: automatic choice.
- 1: sequential analysis. In this case the ordering method is set by ICNTL(7) and the ICNTL(29) (see details below) parameter is meaningless.
- 2: parallel analysis. A parallel ordering and parallel symbolic factorization will be performed if either the PT-SCOTCH or ParMetis parallel ordering tools (or both) are available, depending on the value of ICNTL(29). In this case ICNTL(7) is meaningless and, consequently, all the features accessible through this control parameter are not available.

Any other values will be treated as 0.

At this moment, the parallel analysis is not available for unassembled matrices (i.e., ICNTL(5)=1), in the case where a Schur complement is requested (i.e., ICNTL(19)=1) or in the case where a maximum transversal is requested on the input matrix (i.e., ICNTL(6)=1,2,3,4,5 or 6).

ICNTL(29) is accessed by host process only during the analysis phase and only if a parallel analysis has to be performed, i.e., ICNTL(28)=2 (see details above). It defines the parallel ordering tool to be used to compute the fill-in reducing permutation. Three values are possible:

- 0: automatic choice.
- 1: PT-SCOTCH: the PT-SCOTCH parallel ordering tool will be used to reorder the input matrix, if available.
- 2: ParMetis: the ParMetis parallel ordering tool will be used to reorder the input matrix, if available.

Any other value will be treated as 0. Also, note that ICNTL(29) is meaningless if the sequential analysis is chosen, i.e., ICNTL(28)=1.

ICNTL(30) has default value 0 and is significant only during the solution phase. If ICNTL(30) \neq 0, a user-specified set of entries in the inverse of the original matrix (\mathbf{A}^{-1}) will be computed.

When `ICNTL(30) ≠ 0` then, on entry to the solution phase, the sparse right-hand-side (`NZ_RHS`, `NRHS`, `RHS_SPARSE`, `IRHS_SPARSE`, `IRHS_PTR`, see Section 4.13) should be set to hold the target entries of \mathbf{A}^{-1} that need be computed. `RHS_SPARSE` must be allocated but needs not be initialized and `NRHS` must be set to `N`. On output `RHS_SPARSE`, `IRHS_SPARSE`, `IRHS_PTR` then hold the requested entries of \mathbf{A}^{-1} and `RHS` needs not be allocated by the user.

When a set of entries of \mathbf{A}^{-1} is requested, the associated set of columns will be computed in blocks of size `ICNTL(27)`. In an out-of-core context (`ICNTL(22)=1`), larger `ICNTL(27)` values will most likely decrease the amount of factors read from the disk and reduce the solution time [32, 2]. In an in-core context, the effects might be mixed.

When this functionality is requested, error analysis and iterative refinement will not be performed, even if the corresponding options are set (`ICNTL(10)` and `ICNTL(20)`). Because the entries of \mathbf{A}^{-1} are returned in `RHS_SPARSE` on the host, this functionality is incompatible with the distributed solution option (`ICNTL(21)`). Furthermore, computing entries of \mathbf{A}^{-1} is not possible in the case of partial factorizations with a Schur complement (`ICNTL(19)`).

`ICNTL(31)` has default value 0 and is only accessed by the host during the analysis phase. It is used to indicate that factors may be discarded during the factorization because the solve phase will not require them. `ICNTL(31)` may have the following values:

- 0 (default): all factors needed to perform the solution phase will be kept during the factorization phase.
- 1: indicates that the user is not interested in solving the linear system (Equations 3 or 4) and will not call MUMPS solution phase (`JOB=3`). This option is meaningful when only statistics from the factorization, such as (for example) definiteness, value of the determinant, number of entries in factors after numerical pivoting, number of negative or null pivots are required. In that case, the memory allocated for the factorization will rely on the out-of-core estimates.

Out-of-range values are treated as 0.

`ICNTL(32)` is not used in the current version.

`ICNTL(33)` has default value 0 and is only accessed by the host during the factorization phase. If `ICNTL(33)` is different from 0, the package will attempt to compute the determinant of the input matrix. Note that null pivots (see `ICNTL(24)`), static pivots (see `CNTL(4)`) and elements of the Schur complement (see `ICNTL(18)`) are excluded from the computation of the determinant. To avoid underflows and overflows, the mantissa and the exponent of the determinant are computed separately. The sign of the determinant should be correct for symmetric matrices. When `ICNTL(33)` is different from 0, the determinant is returned in `RINFOG(12)`, `RINFOG(13)`, and `INFOG(34)` such that the determinant is obtained by multiplying (`RINFOG(12)`, `RINFOG(13)`) by 2 to the power `INFOG(34)`. In real arithmetic `RINFOG(13)` is equal to 0.

`ICNTL(34-40)` are not used in the current version.

5.2 Real/complex control parameters

`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 is only accessed by the host during the factorization phase. 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. On unsymmetric or general symmetric matrices, `CNTL(1)` has default value 0.01. For symmetric positive definite matrices numerical pivoting is suppressed and the default value is 0.0. Values less than 0.0 are treated as 0.0. In the unsymmetric case (respectively symmetric case), values greater than 1.0 (respectively 0.5) are treated as 1.0 (respectively 0.5).

CNTL(2) is the stopping criterion for iterative refinement and is only accessed by the host during the solve phase. Let $\mathbf{B}_{\text{err}} = \max_i \frac{|r_i|}{(|\mathbf{A}|_x + |\mathbf{b}|)_i}$ [12]. Iterative refinement will stop when either the required accuracy is reached ($\mathbf{B}_{\text{err}} < \text{CNTL}(2)$) or the convergence rate is too slow (\mathbf{B}_{err} does not decrease by at least a factor of 5). Default value is $\sqrt{\epsilon}$ where ϵ holds the machine precision and depends on the arithmetic version.

CNTL(3) is only used combined with null pivot detection (ICNTL(24) = 1) and is not used otherwise. CNTL(3) has default value 0.0 and is only accessed by the host during the numerical factorization phase. Let A_{pre} be the preprocessed matrix to be factored (see Equation 5). A pivot is considered to be null if the infinite norm of its row/column is smaller than a threshold *thres*. Let ϵ be the machine precision and $\|\cdot\|$ be the infinite norm.

- If CNTL(3) > 0 then $\text{thres} = \text{CNTL}(3) \times \|\mathbf{A}_{\text{pre}}\|$
- If CNTL(3) = 0.0 then $\text{thres} = \epsilon \times 10^{-5} \times \|\mathbf{A}_{\text{pre}}\|$
- If CNTL(3) < 0 then $\text{thres} = |\text{CNTL}(3)|$

CNTL(4) determines the threshold for static pivoting. It is only accessed by the host, and must be set either before the factorization phase, or before the analysis phase. It has default value -1.0. If CNTL(4) < 0.0 static pivoting is not activated. If CNTL(4) > 0.0 static pivoting is activated and the magnitude of small pivots smaller than CNTL(4) will be set to CNTL(4). If CNTL(4) = 0.0 static pivoting is activated and the threshold value used is determined automatically.

CNTL(5) defines the fixation for null pivots and is effective only when null pivot detection is active (ICNTL(24) = 1). CNTL(5) has default value 0.0 and is only accessed by the host during the numerical factorization phase. Let \mathbf{A}_{pre} be the preprocessed matrix to be factored (see Equation 5). If CNTL(5) > 0, in which case we recommend setting it to a large floating-point value in order to limit the impact of this pivot on the rest of the matrix, the detected null pivot is set to +/- CNTL(5) $\times \|\mathbf{A}_{\text{pre}}\|$. The sign of the pivot is preserved in the modified diagonal entry and the factorization continues. In the symmetric case (SYM = 2), if CNTL(5) \leq 0 then the pivot column of the \mathbf{L} factors is set to zero and the pivot entry in matrix \mathbf{D} is set to one. In the unsymmetric case, a large fixation is automatically chosen when CNTL(5) \leq 0 and the remaining part of the pivot row of the \mathbf{U} factor is set to 0.

CNTL(6-15) are not used in the current version.

5.3 Compatibility between options

As shown above, the package has a lot of options and this gives an exponential amount of combinations of options. Almost all options are indeed compatible with each other but obviously a few of them are not, either because the implementation of some options is more complicated in some context, or because some algorithms cannot be applied or do not make sense under certain conditions. For each option and ICNTL parameter, the list of incompatibilities is normally given in the description of the option. The objective of this section is to provide to the user a more global view of the main incompatibilities.

Table 1 highlights the incompatibilities between functionalities and matrix input formats (functionalities which do not appear in this table are compatible with all matrix input formats).

In Table 2, we present the numerical limitations of the solver when ScaLAPACK is used on the final dense Schur complement of MUMPS.

Regarding the solve phase (JOB=3), iterative refinement and error analysis are incompatible with some options, as reported in Table 3. Although iterative refinement and error analysis could be performed externally to the package, they are provided by the package for convenience for all matrix formats but not for all situations. For example, they do not really make sense when computing something different from the solution of $Ax = b$ (e.g. entries of the inverse, null space basis, only forward substitution performed).

Finally, note that orderings available for the sequential and the parallel analysis phase (see ICNTL(28)) are controlled by two different parameters (ICNTL(7) and ICNTL(29)), which have a different range of allowed values, so their is no incompatibility as such. But orderings based on minimum-degree (for example) are only available with the sequential analysis.

Functionality	(Control)	Matrix input format (ICNTL(18) and ICNTL(5))		
		Centralised		Distributed assembled (distr. elemental not avail.)
		Assembled	Elemental	
Unsymmetric permutations	(ICNTL(6))	All options	Not available (ICNTL(6)=0)	Not available (ICNTL(6)=0)
Scalings	(ICNTL(8))	All options	Only option 1 (user-provided)	Only options 7, 8, or 1 (user-provided)
Constrained/compressed orderings	(ICNTL(12))	All options	Not available (ICNTL(12)=0)	Not available (ICNTL(12)=0)
Type of analysis	(ICNTL(28))	Sequential (parallel not available)		Sequential or parallel
Schur complement	(ICNTL(19))	All options		All options but not compatible with Parallel analysis

Table 1: Compatibilities between MUMPS functionalities and matrix-input formats.

	SCALAPACK	
	OFF	ON
Null pivot list (ICNTL(24))	ok	null pivots on root node not available and failure if exact null pivot on root
LDL^t factorization (SYM=2)	ok	ok but LU /PDGETRF performed on root node (no Scalapack LDL^t kernel)
Number of negative pivots (INFOG(12))	ok	lowerbound (negative pivots not counted on root node)

Table 2: MUMPS relies on ScaLAPACK to factorize the last dense Schur complement. If exact inertia (number of negative pivots) or null pivot list is critical, ScaLAPACK can be switched off, see ICNTL(13) although this might imply a small performance degradation.

Functionality	Control	iterative refinement ICNTL(10)	error analysis ICNTL(11)
Multiple right-hand sides	NRHS > 1	Incomp.	Incomp.
Distributed solution	ICNTL(21)	Incomp.	Incomp.
Reduced right-hand sides/ partial solution	ICNTL(26)=1 ICNTL(26)=2	Incomp. Incomp.	Incomp. Incomp.
Compute null space (*)	ICNTL(25)	Incomp.	Incomp.
Entries of A^{-1}	ICNTL(30)	Incomp.	Incomp.

Table 3: List of incompatibilities with postprocessings options at the end of the solve phase. (*) The null space estimate is only available for symmetric matrices.

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 $\text{INFO}(1) = -1$, INFO(2) is the processor number (in communicator `mumps_par%COMM`) on which the error was detected.

INFO(3) - after analysis: Estimated size of the real/complex space needed on the processor to store the factors in memory if the factorization is performed in-core ($\text{ICNTL}(22)=0$). If INFO(3) is negative, then the absolute value corresponds to *millions* of real/complex entries used to store the factor matrices. If the user plans to perform an out-of-core factorization ($\text{ICNTL}(22)=1$), then a rough estimation of the size of the disk space in bytes of the files written by the concerned processor can be obtained by multiplying INFO(3) (or its absolute value multiplied by 1 million when negative) by 4, 8, 8, or 16 for single precision, double precision, single complex, and double complex arithmetics, respectively. The effective value will be returned in INFO(9) (see below), but only after the factorization.

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 estimated size of the main internal integer workarray IS to run the numerical factorization *in-core*.

INFO(8) - after analysis: Minimum estimated size of the main internal real/complex workarray S to run the numerical factorization *in-core*. If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray.

INFO(9) - after factorization: Size of the real/complex space used on the processor to store the factor matrices. If negative, then the absolute value corresponds to *millions* of real/complex entries used to store the factor matrices. In the case of an out-of-core execution ($\text{ICNTL}(22)=1$), the disk space in bytes of the files written by the concerned processor can be obtained by multiplying INFO(9) (or its absolute value multiplied by 1 million) by 4, 8, 8, or 16 for single precision, double precision, single complex, and double complex arithmetics, respectively.

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

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

INFO(12) - after factorization: Number of off-diagonal pivots selected 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. (This means that the user should set `ICNTL(13)=1` or use a single processor in order to get the exact number of off-diagonal or negative pivots rather than a lower bound.) Note that for complex symmetric matrices (`SYM=1` or `2`), `INFO(12)` will be 0. See also `INFOG(12)`, which provides the total number of off-diagonal or negative pivots over all processors.

INFO(13) - after factorization: The number of postponed elimination because of numerical issues.

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

INFO(15) - after analysis: estimated size in Megabytes of all working space to run the numerical phases (factorisation/solve) `in-core` (`ICNTL(22)=0` for the factorization). The maximum and sum over all processors are returned respectively in `INFOG(16)` and `INFOG(17)`.

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

INFO(17) - after analysis: estimated size in Megabytes of all working space to run the numerical phases `out-of-core` (`ICNTL(22)≠0`) with the default strategy. The maximum and sum over all processors are returned respectively in `INFOG(26)` and `INFOG(27)`.

INFO(18) - after factorization: local number of null pivots resulting from detected when `ICNTL(24)=1`.

INFO(19) - after analysis: Estimated size of the main internal integer workarray IS to run the numerical factorization `out-of-core`.

INFO(20) - after analysis: Estimated size of the main internal real/complex workarray S to run the numerical factorization `out-of-core`. If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray.

INFO(21) - after factorization: Effective space used in the main real/complex workarray S. If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray.

INFO(22) - after factorization: Size in millions of bytes of memory effectively used during factorization.

INFO(23) - after factorization: total number of pivots eliminated on the processor. In the case of a distributed solution (see `ICNTL(21)`), this should be used by the user to allocate solution vectors `ISOL_loc` and `SOL_loc` of appropriate dimensions (`ISOL_loc` of size `INFO(23)`, `SOL_loc` of size `LSOL_loc × NRHS` where `LSOL_loc ≥ INFO(23)`) on that processor, between the factorization and solve steps.

INFO(24) - after analysis: estimated number of entries in factors on the processor. If negative, then the absolute value corresponds to *millions* of entries in the factors. Note that in the unsymmetric case, `INFO(24)=INFO(3)`. In the symmetric case, however, `INFO(24) < INFO(3)`.

INFO(25) - After factorization : number of tiny pivots (number of pivots modified by static pivoting) detected on the processor.

INFO(26) - after solution: effective size in Megabytes of all working space to run the solution phase. (The maximum and sum over all processors are returned respectively in `INFOG(30)` and `INFOG(31)`).

INFO(27) - after factorization: effective number of entries in factors on the processor. If negative, then the absolute value corresponds to *millions* of entries in the factors. Note that in the unsymmetric case, `INFO(27)=INFO(9)`. In the symmetric case, however, `INFO(27) ≤ INFO(9)`. The total number of entries over all processors is available in `INFOG(29)`.

INFO(28) - `INFO(40)` are not used in the current version.

6.2 Information available on all processors

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 if `ICNTL(11) ≠ 0`. See description of `ICNTL(11)`.

RINFOG(12) - after factorization: if the computation of the determinant was requested (see `ICNTL(33)`), RINFOG(12) contains the real part of the determinant. The determinant may contain an imaginary part in case of complex arithmetic (see `RINFOG(13)`). It is obtained by multiplying (`RINFOG(12)`, `RINFOG(13)`) by 2 to the power `INFOG(34)`.

RINFOG(13) - after factorization: if the computation of the determinant was requested (see `ICNTL(33)`), RINFOG(13) contains the imaginary part of the determinant. The determinant is then obtained by multiplying (`RINFOG(12)`, `RINFOG(13)`) by 2 to the power `INFOG(34)`.

RINFOG(14) - 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 identical 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`, and with `INFO(1)=-1` and `INFO(2)=p`.

INFOG(3) - after analysis: Total (sum over all processors) estimated real/complex workspace to store the factor matrices. If negative, then the absolute value corresponds to *millions* of real/complex entries used to store the factor matrices. If the user plans to perform an out-of-core factorization (`ICNTL(22)=1`), then a rough estimate of the total disk space in bytes (for all processors) can be obtained by multiplying `INFOG(3)` (or its absolute value multiplied by 1 million) by 4, 8, 8, or 16 for single precision, double precision, single complex, and double complex arithmetics, respectively. The effective value is returned in `INFOG(9)` (see below), but only after the factorization.

INFOG(4) - after analysis: Total (sum over all processors) estimated integer workspace to store the factor matrices

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: the ordering method actually used. The returned value will depend on the type of analysis performed, e.g. sequential or parallel (see `INFOG(32)`). Please refer to `ICNTL(7)` and `ICNTL(29)` for more details on the ordering methods available in sequential and parallel analysis respectively.

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 which will be the case if the input matrix is in elemental form.)

INFOG(9) - after factorization: Total (sum over all processors) real/complex workspace to store the factor matrices. If negative, then the absolute value corresponds to the size in *millions* of real/complex entries used to store the factors. In case of an out-of-core factorization (**ICNTL(22)**=1, the total disk space in bytes of the files written by all processors can be obtained by multiplying **INFOG(9)** (or its absolute value multiplied by 1 million) by 4, 8, 8, or 16 for single precision, double precision, single complex, and double complex arithmetics, respectively.

INFOG(10) - after factorization: Total (sum over all processors) integer workspace to store the factor matrices.

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 (real arithmetic) if **SYM**=1 or 2. If **ICNTL(13)**=0 (the default) this excludes pivots from the parallel root node treated by ScaLAPACK. (This means that the user should set **ICNTL(13)** to a positive value, say 1, or use a single processor in order to get the exact number of off-diagonal or negative pivots rather than a lower bound.) Furthermore, when **ICNTL(24)** is set to 1 and **SYM**=1 or 2, **INFOG(12)** excludes the null⁷ pivots, even if their sign is negative. In other words, a pivot cannot be both null and negative.

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. A large number (more than 10% of the order of the matrix) indicates numerical problems. Settings related to numerical preprocessing (**ICNTL(6)**,**ICNTL(8)**, **ICNTL(12)**) might then be modified by the user.

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

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

INFOG(16) and INFOG(17) - after analysis: Estimated size (in million of bytes) of all MUMPS internal data for running factorization *in core*.

- —(16) : value on the most memory consuming processor.
- —(17) : sum over all processors.

INFOG(18) and INFOG(19) - after factorization: Size in millions of bytes of all MUMPS internal data allocated during factorization.

- —(18) : value on the most memory consuming processor.
- —(19) : sum over all processors.

INFOG(20) - after analysis: Estimated number of entries in the factors. If negative the absolute value corresponds to *millions* of entries in the factors. Note that in the unsymmetric case, **INFOG(20)**=**INFOG(3)**. In the symmetric case, however, **INFOG(20)** < **INFOG(3)**.

INFOG(21) and INFOG(22) - after factorization: Size in millions of bytes of memory effectively used during factorization.

- —(21) : value on the most memory consuming processor.
- —(22) : sum over all processors.

INFOG(23) - After analysis: value of **ICNTL(6)** effectively used.

INFOG(24) - After analysis: value of **ICNTL(12)** effectively used.

INFOG(25) - After factorization : number of tiny pivots (number of pivots modified by static pivoting)

INFOG(26) and INFOG(27) - after analysis: Estimated size (in millions of bytes) of all MUMPS internal data for running factorization *out-of-core* (**ICNTL(22)**≠ 0) for a given value of **ICNTL(14)** and for the default strategy.

- —(26) : max over all processors
- —(27) : sum over all processors

INFOG(28) - After factorization: number of null pivots encountered. See **CNTL(3)** for the definition of a null pivot.

⁷i.e., whose magnitude is smaller than the tolerance defined by **CNTL(3)**.

INFOG(29) - After factorization: effective number of entries in the factors (sum over all processors). If negative, then the absolute value corresponds to *millions* of entries in the factors. Note that in the unsymmetric case, INFOG(29)=INFOG(9). In the symmetric case, however, INFOG(29) ≤ INFOG(9).

INFOG(30) and INFOG(31) - after solution: Size in millions of bytes of memory effectively used during solution phase:

- —(30) : max over all processors
- —(31) : sum over all processors

INFOG(32) - after analysis: the type of analysis actually done (see ICNTL(28)). INFOG(32) has value 1 if sequential analysis was performed, in which case INFOG(7) returns the sequential ordering option used, as defined by ICNTL(7). INFOG(32) has value 2 if parallel analysis was performed, in which case INFOG(7) returns the parallel ordering used, as defined by ICNTL(29).

INFOG(33): effective value used for ICNTL(8). It is set both after the analysis and the factorization phases. If ICNTL(8)=77 on entry to the analysis and INFOG(33) has value 77 on exit from the analysis, then no scaling was computed during the analysis and the automatic decision will only be done during factorization (except if the user modifies ICNTL(8) to set a specific option on entry to the factorization).

INFOG(34): if the computation of the determinant was requested (see ICNTL(33)), INFOG(34) contains the exponent of the determinant. See also RINFOG(12) and RINFOG(13): the determinant is obtained by multiplying (RINFOG(12), RINFOG(13)) by 2 to the power INFOG(34).

INFOG(35) - 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 the 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 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*). If several processors raised an error, those processors 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 rank of the processor having the most negative error code.

The behaviour is slightly different for the global information parameters `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`), or the initialization phase (`JOB=-1`) was performed a second time on an instance not freed (`JOB=-2`). See description of `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` at position `INFO(2)`. This error may only occur on the host.

- 5 Problem of REAL or COMPLEX workspace allocation of size **INFO(2)** during analysis.
- 6 Matrix is singular in structure. **INFO(2)** holds the structural rank.
- 7 Problem of INTEGER workspace allocation of size **INFO(2)** during analysis.
- 8 Main internal integer workarray IS 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)** before calling the factorization again (**JOB=2**).
- 9 Main internal real/complex workarray S too small. If **INFO(2)** is positive, then the number of entries that are missing in S at the moment when the error is raised is available in **INFO(2)**. If **INFO(2)** is negative, then its absolute value should be multiplied by 1 million. If an error -9 occurs, the user should increase the value of **ICNTL(14)** before calling the factorization (**JOB=2**) again, except if **ICNTL(23)** is provided, in which case **ICNTL(23)** should be increased.
- 10 Numerically singular matrix.
- 11 Internal real/complex workarray S too small for solution. Please contact us. If **INFO(2)** is positive, then the number of entries that are missing in S at the moment when the error is raised is available in **INFO(2)**.
- 12 Internal real/complex workarray S too small for iterative refinement. Please contact us.
- 13 An error occurred in a Fortran ALLOCATE statement. The size that the package requested is available in **INFO(2)**. If **INFO(2)** is negative, then the size that the package requested is obtained by multiplying the absolute value of **INFO(2)** by 1 million.
- 14 Internal integer workarray IS too small for solution. See error **INFO(1) = -8**.
- 15 Integer workarray IS 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 calling MUMPS again.
- 20 The internal reception buffer that was allocated dynamically by MUMPS 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 calling MUMPS again.
- 21 Value of **PAR=0** is not allowed because only one processor is available; 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 insufficient size, or
 - is associated and should not be associated (for example, RHS on non-host processors).

INFO(2) points to the incorrect pointer array 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
10	RHS_SPARSE
11	IRHS_SPARSE
12	IRHS_PTR
13	ISOL_loc
14	SOL_loc
15	REDRHS

- 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 occurred 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.
- 26 `LRHS` is out of range. `INFO(2)=LRHS`.
- 27 `NZ_RHS` and `IRHS_PTR(NRHS+1)` do not match. `INFO(2) = IRHS_PTR(NRHS+1)`.
- 28 `IRHS_PTR(1)` is not equal to 1. `INFO(2) = IRHS_PTR(1)`.
- 29 `LSOL_loc` is smaller than `INFO(23)`. `INFO(2)=LSOL_loc`.
- 30 `SCHUR_LLD` is out of range. `INFO(2) = SCHUR_LLD`.
- 31 A 2D block cyclic symmetric (`SYM=1` or `2`) Schur complement is required with the option `ICNTL(19)=3`, but the user has provided a process grid that does not satisfy the constraint `MBLOCK=NBLOCK`. `INFO(2)=MBLOCK-NBLOCK`.
- 32 Incompatible values of `NRHS` and `ICNTL(25)`. Either `ICNTL(25)` was set to `-1` and `NRHS` is different from `INFO(28)`; or `ICNTL(25)` was set to i , $1 \leq i \leq \text{INFO}(28)$ and `NRHS` is different from 1. Value of `NRHS` is stored in `INFO(2)`.
- 33 `ICNTL(26)` was asked for during solve phase but the Schur complement was not asked for at the analysis phase (`ICNTL(19)`). `INFO(2)=ICNTL(26)`.
- 34 `LREDRHS` is out of range. `INFO(2)=LREDRHS`.
- 35 This error is raised when the expansion phase is called (`ICNTL(26) = 2`) but reduction phase (`ICNTL(26)=1`) was not called before. `INFO(2)` contains the value of `ICNTL(26)`.
- 36 Incompatible values of `ICNTL(25)` and `INFO(28)`. The value of `ICNTL(25)` is stored in `INFO(2)`.
- 37 Value of `ICNTL(25)` incompatible with some other parameter. with `SYM` or `ICNTL(xx)`. If `INFO(2)=0` then `ICNTL(25)` is incompatible with `SYM`: in current version, the null space basis functionality is not available for unsymmetric matrices (`SYM=0`). Otherwise, `ICNTL(25)` is incompatible with `ICNTL(xx)`, and the index `xx` is stored in `INFO(2)`.
- 38 Parallel analysis was set (i.e., `ICNTL(28)=2`) but PT-SCOTCH or ParMetis were not provided.
- 39 Incompatible values for `ICNTL(28)` and `ICNTL(5)` and/or `ICNTL(19)` and/or `ICNTL(6)`. Parallel analysis is not possible in the cases where the matrix is unassembled and/or a Schur complement is requested and/or a maximum transversal is requested on the matrix.
- 40 The matrix was indicated to be positive definite (`SYM=1`) by the user but a negative or null pivot was encountered during the processing of the root by ScaLAPACK. `SYM=2` should be used.
- 44 The solve phase (`JOB=3`) cannot be performed because the factors or part of the factors are not available. `INFO(2)` contains the value of `ICNTL(31)`.
- 45 $\text{NRHS} \leq 0$. `INFO(2)` contains the value of `NRHS`.
- 46 $\text{NZ_RHS} \leq 0$. This is currently not allowed in case of reduced right-hand-side (`ICNTL(26)=1`) and in case entries of \mathbf{A}^{-1} are requested (`ICNTL(30)=1`). `INFO(2)` contains the value of `NZ_RHS`.
- 47 Entries of \mathbf{A}^{-1} were requested during the solve phase (`JOB=3`, `ICNTL(30)=1`) but the constraint `NRHS=N` is not respected. The value of `NRHS` is provided in `INFO(2)`.
- 48 \mathbf{A}^{-1} Incompatible values of `ICNTL(30)` and `ICNTL(xx)`. `xx` is stored in `INFO(2)`.
- 49 `SIZE_SCHUR` has an incorrect value (`SIZE_SCHUR < 0` or `SIZE_SCHUR ≥ N`, or `SIZE_SCHUR` was modified on the host since the analysis phase. The value of `SIZE_SCHUR` is provided in `INFO(2)`.
- 90 Error in out-of-core management. See the error message returned on output unit `ICNTL(1)` for more information.

A positive value of `INFO(1)` is associated with a warning message which will be output on unit `ICNTL(2)` when `ICNTL(4) ≥ 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.
- +4 User data `JCN` has been modified (internally) by the solver.
- +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 4 and 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 arithmetic with the following prototype:

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

An example of calling MUMPS from C for a complex assembled problem is given in Section 10.3. 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:

```
#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 $\{1, 2, \dots, NZ\}$ 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 4.6), 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, and most MPI implementations, you may just do

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

```

typedef struct
{
  int sym, par, job;
  int comm_fortran; /* Fortran communicator */
  int icntl[40];
  real cntl[15];
  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;
  /* RHS, solution, output data and statistics */
  real/complex *rhs, *redrhs, *rhs_sparse, *sol_loc;
  int *irhs_sparse, *irhs_ptr, *isol_loc;
  int nrhs, lrhs, lredrhs, nz_rhs, lsol_loc;
  int info[40], infog[40];
  real rinfo[20], rinfog[20];
  int *sym_perm, *uns_perm;
  int * mapping;
  /* Schur */
  int size_schur; int *listvar_schur; real/complex *schur;
  int nprow, npcol, mblock, nblock, schur_lld, schur_mloc, schur_nloc;
  /* Version number */
  char version_number[80];
  char ooc_tmpdir[256], ooc_prefix[64];          char write_problem[256];
  /* 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.

(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) comm_c;
```

should work.

For some MPI implementations, check if `id.comm_fortran = MPIR_FromPointer(comm_c)` can be used.

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 were 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 2.7.

9 Scilab and MATLAB/Octave interfaces

Thanks to Octave MEX compatibility, an Octave interface can be generated based on the MATLAB one. All the documentation provided in this section for the MATLAB interface, also applies to the Octave case.

The main callable functions are

```

id = initmumps;
id = dmumps(id [,mat] );
id = zmumps(id [,mat] );

```

We have designed these interfaces such that their usage is as similar as possible to the existing C and Fortran interfaces to MUMPS. Only an interface to the sequential version of MUMPS is provided, thus only the parameters related to the sequential version of MUMPS are available. (Note that in the out-of-core case, functionalities allowing to control the directory and name of temporary files, are, however, not currently available.) The main differences and characteristics are:

- The existence of a function `initmumps` (usage: `id=initmumps`) that builds an initial structure `id` in which `id.JOB` is set to -1 and `id.SYM` is set to 0 (unsymmetric solver by default).
- Only the double precision and double complex versions of MUMPS are interfaced, since they correspond to the arithmetics used in MATLAB/Scilab.
- the sparse matrix A is passed to the interface functions `dmumps` and `zmumps` as a Scilab/MATLAB object (parameters `ICNTL(5)`, `N`, `NZ`, `NELT`, ... are thus irrelevant).
- the right-hand side vector or matrix, possibly sparse, is passed to the interface functions `dmumps` and/or `zmumps` in the argument `id.RHS`, as a Scilab/MATLAB object (parameters `ICNTL(20)`, `NRHS`, `NZRHS`, ... are thus irrelevant).
- The Schur complement matrix, if required, is allocated within the interface and returned as a Scilab/MATLAB dense matrix. Furthermore, the parameters `SIZE_SCHUR` and `ICNTL(19)` need not be set by the user; they are set automatically depending on the availability and size of the list of Schur variables, `id.VAR_SCHUR`.
- We have chosen to use a new variable `id.SOL` to store the solution, instead of overwriting `id.RHS`.

Please refer to the report [22] for a more detailed description of these interfaces. Please also refer to the README file in directories MATLAB or Scilab of the main MUMPS distribution for more information on installation. For example, one important thing to note is that at installation, the user must provide the Fortran 90 runtime libraries corresponding to the compiled MUMPS package. This can be done in the makefile for the MATLAB interface (file `make.inc`) and in the builder for the Scilab interface (file `builder.sce`).

Finally, note that examples of usage of the MATLAB and the Scilab interfaces are provided in directories MATLAB and Scilab/examples, respectively. In the following, we describe the input and output parameters of the function `[dz]mumps`, that are relevant in the context of this interface to the sequential version of MUMPS.

Input Parameters

- **mat** : sparse matrix which has to be provided as the second argument of `dmumps` if `id.JOB` is strictly larger than 0.
- **id.SYM** : controls the matrix type (symmetric positive definite, symmetric indefinite or unsymmetric) and it has to be initialized by the user before the initialization phase of MUMPS (see `id.JOB`). Its value is set to 0 after the call of `initmumps`.
- **id.JOB** : defines the action that will be realized by MUMPS: initialize, analyze and/or factorize and/or solve and release MUMPS internal C/Fortran data. It has to be set by the user before any call to MUMPS (except after a call to `initmumps`, which sets its value to -1).
- **id.ICNTL and id.CNTL** : define control parameters that can be set after the initialization call (`id.JOB = -1`). See Section "Control parameters" for more details. If the user does not modify an entry in `id.ICNTL` then MUMPS uses the default parameter. For example, if the user wants to use the AMD ordering, he/she should set `id.ICNTL(7) = 0`. Note that the following parameters are inhibited because they are automatically set within the interface: `id.ICNTL(19)` which controls the Schur complement option and `id.ICNTL(20)` which controls the format of the right-hand side. Note that parameters `id.ICNTL(1:4)` may not work properly depending on your compiler and your environment. In case of problem, we recommend to switch printing off by setting `id.ICNTL(1:4)=-1`.

- **id.PERM_IN** : corresponds to the given ordering option (see Section “Input and output parameters” for more details). Note that this permutation is only accessed if the parameter `id.ICNTL(7)` is set to 1.
- **id.COLSCA** and **id.ROWSCA** : are optional scaling arrays (see Section “Input and output parameters” for more details)
- **id.RHS** : defines the right-hand side. The parameter `id.ICNTL(20)` related to its format (sparse or dense) is automatically set within the interface. Note that `id.RHS` is not modified (as in MUMPS), the solution is returned in `id.SOL`.
- **id.VAR_SCHUR** : corresponds to the list of variables that appear in the Schur complement matrix (see Section “Input and output parameters” for more details).
- **id.REDRHS** (input parameter only if `id.VAR_SCHUR` was provided during the factorization and if `ICNTL(26)=2` on entry to the solve phase): partial solution on the variables corresponding to the Schur complement. It is provided by the user and normally results from both the Schur complement and the reduced right-hand side that were returned by MUMPS in a previous call. When `ICNTL(26)=2`, MUMPS uses this information to build the solution `id.SOL` on the complete problem. See Section “Schur complement” for more details.

Output Parameters

- **id.SCHUR** : if `id.VAR_SCHUR` is provided of size `SIZE_SCHUR`, then `id.SCHUR` corresponds to a dense array of size `(SIZE_SCHUR,SIZE_SCHUR)` that holds the Schur complement matrix (see Section “Input and output parameters” for more details). The user does not have to initialize it.
- **id.REDRHS** (output parameter only if `ICNTL(26)=1` and `id.VAR_SCHUR` was defined): Reduced right-hand side (or condensed right-hand side on the variables associated to the Schur complement). It is computed by MUMPS during the solve stage if `ICNTL(26)=1`. It can then be used outside MUMPS, together with the Schur complement, to build a solution on the interface. See Section “Schur complement” for more details.
- **id.INFOG** and **id.RINFOG** : information parameters (see Section “Information parameters”).
- **id.SYM_PERM** : corresponds to a symmetric permutation of the variables (see discussion regarding `ICNTL(7)` in Section “Control parameters”). This permutation is computed during the analysis and is followed by the numerical factorization except when numerical pivoting occurs.
- **id.UNS_PERM** : column permutation (if any) on exit from the analysis phase of MUMPS (see discussion regarding `ICNTL(6)` in Section “Control parameters”).
- **id.SOL** : dense vector or matrix containing the solution after MUMPS solution phase.

Internal Parameters

- `id.INST`: (MUMPS reserved component) MUMPS internal parameter.
- `id.TYPE`: (MUMPS reserved component) defines the arithmetic (complex or double precision).

10 Examples of use of MUMPS

10.1 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_struct.h` for MUMPS. The file `mumps_root.h` must also be available because it is included in `mumps_struct.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

```

PROGRAM MUMPS_EXAMPLE
  INCLUDE 'mpif.h'
  INCLUDE 'dmumps_struct.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

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

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

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
13.0      : RHS

```

and we obtain the solution $\text{RHS}(i) = i, i = 1, \dots, 5$.

10.2 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 10.1 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

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

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 $\text{RHS}(i) = i, i = 1, \dots, 5$.

```

PROGRAM MUMPS_EXAMPLE
INCLUDE 'mpif.h'
INCLUDE 'dmumps_struct.h'
TYPE (DMUMPS_STRUC) id
INTEGER IERR, LEITVAR, 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,*) LEITVAR
    READ(5,*) NA_ELT
    ALLOCATE( id%ELTPTR ( id%NELT+1 ) )
    ALLOCATE( id%ELTVAR ( LEITVAR ) )
    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, LEITVAR )
    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)
C Deallocate user data
  DEALLOCATE( id%ELTPTR )
  DEALLOCATE( id%ELTVAR )
  DEALLOCATE( id%A_ELT )
  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 4: Example program using MUMPS on an elemental DOUBLE PRECISION problem.

10.3 An example of calling MUMPS from C

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

```

/* 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]^T
 * Solution is [1 2]^T */
#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.

11 Notes on MUMPS distribution

This version of MUMPS is provided to you free of charge. It is public domain, based on public domain software developed during the Esprit IV European project PARASOL (1996-1999). Since this first public domain version in 1999, research and developments have been supported by the following institutions: CERFACS, CNRS, ENS Lyon, INPT(ENSEEIH)-IRIT, INRIA, and University of Bordeaux.

The MUMPS team at the moment of releasing this version includes Patrick Amestoy, Maurice Bremond, Alfredo Buttari, Abdou Guermouche, Guillaume Joslin, Jean-Yves L'Excellent, Francois-Henry Rouet, Bora Ucar and Clement Weisbecker.

We are also grateful to Emmanuel Agullo, Caroline Bousquet, Indranil Chowdhury, Philippe Combes, Christophe Daniel, Iain Duff, Vincent Espirat, Aurelia Fevre, Jacko Koster, Stephane Pralet, Chiara Puglisi, Gregoire Richard, Tzvetomila Slavova, Miroslav Tuma and Christophe Voemel who have been contributing to this project.

Up-to-date copies of the MUMPS package can be obtained from the Web pages:
<http://mumps.enseeiht.fr/> or <http://graal.ens-lyon.fr/MUMPS>

THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED OR IMPLIED. ANY USE IS AT YOUR OWN RISK.

User documentation of any code that uses this software can include this complete notice. You can acknowledge (using references [1] and [2]) the contribution of this package in any scientific publication dependent upon the use of the package. You shall use reasonable endeavours to notify the authors of the package of this publication.

[1] P. R. Amestoy, I. S. Duff, J. Koster and J.-Y. L'Excellent, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal of Matrix Analysis and Applications, Vol 23, No 1, pp 15-41 (2001).

[2] P. R. Amestoy and A. Guermouche and J.-Y. L'Excellent and S. Pralet, Hybrid scheduling for the parallel solution of linear systems. Parallel Computing Vol 32 (2), pp 136-156 (2006).

Other acknowledgements

Apart from the contributors cited above, we are grateful to Jürgen Schulze for letting us distribute PORD developed at the University of Paderborn, and to Alexis Salzman for his help regarding the determinant.

We thank Eddy Caron for the administration of a server used everyday by the MUMPS team.

We also want to thank BRGM, EADS-CCR, LARIA, Lawrence Berkeley National Laboratory, PARALLAB (Bergen) and Rutherford Appleton Laboratory for research discussions that have certainly influenced this work.

Finally we want to thank the institutions that have provided access to their parallel machines: Centre

Informatique National de l'Enseignement Supérieur (CINES), CERFACS, CICT (Toulouse), Fédération Lyonnaise de Calcul Haute-Performance, Institut du Développement et des Ressources en Informatique Scientifique (IDRIS), Lawrence Berkeley National Laboratory, Laboratoire de l'Informatique du Parallélisme, INRIA Rhones-Alpes, PARALLAB.

References

- [1] E. Agullo. *On the Out-of-core Factorization of Large Sparse Matrices*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.
- [2] P. Amestoy, I. Duff, Y. Robert, F.-H. Rouet, and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. Technical report RT-APO-10-06, University of Toulouse-IRIT, juin 2010. Also appeared as INRIA-LIP technical report.
- [3] 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.
- [4] 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.
- [5] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputer Applications*, 3:41–59, 1989.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] P. R. Amestoy, I. S. Duff, D. Ruiz, and B. Uçar. A parallel matrix scaling algorithm. In *High Performance Computing for Computational Science, VECPAR'08*, number 5336 in Lecture Notes in Computer Science, pages 309–321. Springer-Verlag, 2008.
- [11] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.
- [12] 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.
- [13] 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.
- [14] A. R. Curtis and J. K. Reid. On the automatic scaling of matrices for Gaussian elimination. *J. Inst. Maths. Applics.*, 10:118–124, 1972.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] I. S. Duff and S. Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications*, 27(2):313–340, 2005.

- [20] 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.
- [21] 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.
- [22] A. Fèvre, J.-Y. L’Excellent, and S. Pralet. Scilab and MATLAB interfaces to MUMPS. Technical Report RR-5816, INRIA, Jan. 2006. Also appeared as ENSEEIHT-IRIT report TR/TLSE/06/01 and LIP report RR2006-06.
- [23] A. Guermouche. *Étude et optimisation du comportement mémoire dans les méthodes parallèles de factorisation de matrices creuses*. PhD thesis, École Normale Supérieure de Lyon, July 2004.
- [24] A. Guermouche and J.-Y. L’Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.
- [25] A. Guermouche, J.-Y. L’Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.
- [26] G. Karypis and V. Kumar. *MEÏS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, Sept. 1998.
- [27] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
- [28] F. Pellegrini. SCOTCH 5.0 User’s guide. Technical Report, LaBRI, Université Bordeaux I, Aug. 2007.
- [29] S. Pralet. *Constrained orderings and scheduling for parallel sparse linear algebra*. PhD thesis, Institut National Polytechnique de Toulouse, Sept 2004. Available as CERFACS technical report, TH/PA/04/105.
- [30] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RT/APO/01/4, ENSEEIHT-IRIT, 2001. Also appeared as RAL report RAL-TR-2001-034.
- [31] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.
- [32] Tz. Slavova. *Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems*. Ph.D. dissertation, Institut National Polytechnique de Toulouse, Apr. 2009. Available as CERFACS Report TH/PA/09/59.
- [33] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.