MUltifrontal Massively Parallel Solver
(`MUMPS 5.1.2`)
Users' guide [*]

October 2, 2017

Abstract

This document describes the Fortran 95 and C user interfaces to `MUMPS 5.1.2`. We describe in detail the data structures, parameters, calling sequences, and error diagnostics. Basic example programs using `MUMPS` are also provided.

First improvements regarding low-rank compression (Block-Low Rank, or BLR) format are now available in public releases of MUMPS.

---

[*]Information on how to obtain updated copies of `MUMPS` can be obtained from the Web page http://mumps-solver.org/

Contents

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, on distributed memory computers. MUMPS implements a direct method based on a multifrontal approach which performs a Gaussian 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 [9, 10, 13, 28, 29, 33, 40, 31, 32, 18, 1, 45, 15, 41, 3, 35, 47, 19, 36, 44] for full details of the techniques used, algorithms and related research.

The system $\mathbf{Ax} = \mathbf{b}$ is solved in three main steps:

1. Analysis.
   During analysis, preprocessing (see Subsection 3.2), including an ordering based on the symmetrized pattern $\mathbf{A} + \mathbf{A}^T$, and a symbolic factorization are performed. During **the symbolic factorization**, a mapping of the multifrontal computational graph, the so called **elimination tree** [38], is computed and used to estimate the number of operations and memory necessary for factorization and solution. Both parallel and sequential implementations of the analysis phase are available. Let $\mathbf{A_{pre}}$ denote the preprocessed matrix (further defined in Subsection 3.2).

2. Factorization.
   During factorization $\mathbf{A_{pre}} = \mathbf{LU}$ or $\mathbf{A_{pre}} = \mathbf{LDL}^T$, 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 computed during the analysis. 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 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 x_{pre} of $\mathbf{LUx_{pre}} = \mathbf{b_{pre}}$ or $\mathbf{LDL}^T\mathbf{x_{pre}} = \mathbf{b_{pre}}$ 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

   $$\mathbf{Ly} = \mathbf{b_{pre}} \text{ or } \mathbf{LDy} = \mathbf{b_{pre}}, \tag{3}$$

   followed by a **backward** elimination step

   $$\mathbf{Ux_{pre}} = \mathbf{y} \text{ or } \mathbf{L}^T\mathbf{x_{pre}} = \mathbf{y}. \tag{4}$$

   The solution $\mathbf{x_{pre}}$ is finally postprocessed to obtain the solution $\mathbf{x}$ of the original system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is either assembled on an identified processor (*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 3 phases can be called separately (see Subsection 5.1.1). A special case is the one where the forward elimination step is performed during factorization (see Subsection 3.7), instead of during the solve phase. This allows accessing the $\mathbf{L}$ factors right after they have been computed, with a better locality, and can avoid writing the $\mathbf{L}$ factors to disk in an out-of-core context. In this case (forward elimination during factorization), only the backward elimination is performed during the solution phase .

The software is mainly written in Fortran 95 although a C interface is available (see Section 9). Scilab and MATLAB/Octave interfaces are also available in the case of sequential executions. The parallel

version of `MUMPS` requires MPI [46] for message passing and makes use of the BLAS [23, 24], LAPACK, BLACS, and ScaLAPACK [22] libraries. The sequential version only relies on BLAS and LAPACK.

`MUMPS` exploits both parallelism arising from sparsity in the matrix $\mathbf{A}$ and from dense factorization kernels. It 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 if it is not kept distributed.

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 Subsection 3.10).

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 is used to accommodate numerical pivoting in the factorization and to remap work and data to appropriate processors at execution time. 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.

The main features of the `MUMPS` package include:

- various arithmetics (real or complex, single or double precision)

- input of the matrix in assembled format (distributed or centralized) or elemental format

- sequential or parallel analysis phase

- use of several built-in ordering algorithms, a tight interface to some external ordering packages such as PORD [43], SCOTCH [39] or Metis [34] (strongly recommended), and the possibility for the user to input a given ordering.

- scaling of the original matrix

- out-of-core capability

- detection of null pivots, basic estimate of rank deficiency and computation of a null space basis for symmetric matrices

- computation of a Schur complement matrix

- computation of the determinant

- computation of selected entries of the inverse of $\mathbf{A}$

- exploiting sparsity of the right-hand sides

- forward elimination during factorization

- solution of the transposed system

- error analysis

- iterative refinement

- selective 64-bit integer

- factorization based on Block Low Rank (BLR) format

`MUMPS` is downloaded from the web site many times per day 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 many parallel computers. Please visit our webstite for recommendations, from our users, on how to use the solver on Windows platforms.

2 Notes for users of previous versions of `MUMPS`

The `MUMPS 5.1.2` release has several new features and several bug fixes with respect to the previous major release (see a list of changes in Subsection 2.1). We strongly advise to use the latest version.

In this section, we also describe backward compatibility issues and what should be done if you are using a version of `MUMPS` anterior to `MUMPS 5.1.2` and if you would like to upgrade your application to use `MUMPS 5.1.2`.

We discuss binary compatibility in Subsection 2.2 and how to upgrade between minor versions in Subsection 2.3. We then discuss in Subsection 2.4 and in Subsection 2.5 some minor modifications to control parameters or to interfaces with ordering packages. Such control parameters are normally backward compatible, but it may happen that their range of possible values or their meaning has been slightly modified or extended. Please read this section if you are using an anterior version of MUMPS and want to use MUMPS 5.1.2.

2.1 ChangeLog

```
Changes from 5.1.1 to 5.1.2
* Corrected an overestimation of memory (regression from 5.1.0)
* Corrected/extended WORKAROUNDINTELILP64MPI2INTEGER mechanism (see INSTALL)
* Parallel analysis: fixed a bug, limited number of MPI processes on small
  problems, and reverted to sequential analysis on tiny problems. This is
  to avoid erroneous behavior and failures in the parallel ordering tools.
* Faster BLR clustering on matrices with quasi-dense rows (which are skipped)
* Improved performance of solve phase on very small matrices
* Solve phase with a single MPI process is more thread-safe
* Fixed compilation issue with opensolaris ([SDCZ]MUMPS_TRUNCATED_RRQR)
* Fixed minor bug in BLR factorization (uninitialized timer)
* Corrected minor compiler warnings
* Minor correction to userguide
* Add -DBLR_MT in Intel example Makefile


Changes from 5.1.0 to 5.1.1
* Fix in parallel analysis
* Stabilization of 5.1.0:
  - Improved stability of Block-Low-Rank feature
  - Corrected an incorrect deallocation of POSINRHSCOMP_COL
  - Correction of a case of uninitialized data access in type 2 pivoting
  - Suppressed occasional debug trace "write(6,*) " KEEP265= ", KEEP265"

Changes from 5.0.2 to 5.1.0
* New feature: selective 64-bit integers (introduced only where needed)
  to process matrices with more than 2^{31}-1 entries.
   -mixed 32/64 bit integers for API: NNZ/NNZ_loc 64-bit
     (NZ/NZ_LOC kept temporarily for backward compatibility)
   - both 32 or 64 bit integer versions of external orderings
     (Metis/ParMetis, SCOTCH/pt-SCOTCH, PORD), can be used
   - Error -51 when a 32-bit external ordering is invoked on
     a graph larger than  2^{31}-1
* New feature: (experimental) factorization based on Block-Low-Rank format,
  (ICNTL(35) to activate it and CNTL(7) for low-rank precision)
* Improved performance on numerically hard matrices (LU and LDLt)
* "-DALLOW_NON_INIT" flag has disappeared and needs no longer be used
* Fixed incorrect deallocation in case of JOB=3/ICNTL(26)=1 followed by JOB=2
* Fixed compilation problem with Intel2017 + openMPI in [sdcz]ana_aux_par.F
* Minor correction of memory statistics for solve
* Use 64-bit integers where needed during the solve phase to enable
  large number of right-hand-sides (NRHS) in one block
  (i.e. ICNTL(27)xN can be larger than 2^{31}-1)
* Improved performance of solve phase
* Allow pivoting thresholds CNTL(1) equal to 1.0
* New error -52: when default Fortran integers are 64 bit, external
```

```
  orderings should also have 64-bit default integers
* New error -22, INFO(2)=16 when IRN_loc or JCN_loc not associated
  while ICNTL(18) is set to 3
* Missing O_BINARY flag was added to open binary files on MINGW systems
* New error -53 that could reflect a matrix structure change between
analysis and factorization
```

2.2 Binary compatibility

In general, successive versions of the MUMPS package do not ensure binary compatibility. This means that you should recompile your code if you use a new version of MUMPS, even in case of a minor release (e.g., MUMPS x.y.z to MUMPS x.y.z', see below).

2.3 Upgrading between minor releases

Between minor releases (e.g., MUMPS x.y.z to MUMPS x.y.z'), binary compatibility is not ensured (see Subsection 2.2). Furthermore, the number of source files (and general Makefile) may have changed. However, the interface is fully backward-compatible. Rebuilding the new version and recompiling the source files of your application that include MUMPS include files should thus be enough to use the latest version.

2.4 Upgrading from MUMPS 5.0.2 to MUMPS 5.1.2

2.4.1 Changes on installation issues

Installation issues are described in the file "INSTALL" and in Makefile.inc examples from the "Make.inc/" directory.

1. Since MUMPS 5.1.0, the variables IMETIS and ISCOTCH should always be defined in your Makefile.inc in case (par)Metis or (pt-)SCOTCH are installed (recommended). This is because some functions relying on Metis and/or SCOTCH now require header files from those ordering packages whereas this was not the case in MUMPS 5.0.x versions.

2. Since MUMPS 5.1.0, LAPACK is required at installation time, even in the case of the MPI-free version. This is because the block-low rank feature uses some LAPACK routines. In MUMPS 5.0.x versions, LAPACK was only necessary as a dependency on ScaLAPACK, in case the ScaLAPACK library used did not come with all its LAPACK dependencies.

2.4.2 Selective 64-bit integer feature

Since MUMPS 5.1.0, in order to process matrices with a large number of entries and/or with a large number of right-hand sides, 64-bit integers have been introduced where needed (and only where needed). With *large number* we mean a number greater than $2^{31} - 1$ (32-bit integer limitation). This has the following implications.

- *Large number of matrix entries (Subsection 5.2.2)*. There are both 32 and 64 bit integers in the MUMPS user interface for the number of entries in the matrix:
  - centralized/distributed input matrix format: the variables NZ/NZ_loc are still 32-bit for backward compatibility and will become obsolete in some future release;
  - centralized/distributed input matrix format: two new variables have been introduced, NNZ/NNZ_loc, which are 64-bit integers (recommended choice).
- *External orderings*. Either the 32 or 64 bit integer versions of external orderings (Metis/ParMetis, SCOTCH/pt-SCOTCH, PORD) can be used (see "INSTALL").
- *External libraries*. 32-bit integers are still used for BLAS, LAPACK, MPI, ScaLAPACK (MPI message doesn't have a count larger than $2^{31} - 1$).

- *-51 error message*. Error -51 which was previously raised in case of integer overflow during analysis is now only raised when a 32-bit external ordering is invoked on a graph with more than $2^{31} - 1$ edges.
- *Large number of right-hand sides*. Large numbers of right-hand sides (`NRHS`) can be processed in a single block (i.e. `ICNTL(27)` $\times$ N is now allowed to be larger than $2^{31} - 1$). However, in order to avoid integer overflows, `ICNTL(27)` should still be small enough for `ICNTL(27)` $\times$ `INFOG(11)` to be smaller than 2^{31}, where `INFOG(11)` $\leq$ N is the maximum frontal size.

With the selective 64-bit integer feature, most `MUMPS` integer arrays will remain 32-bit integers, since `MUMPS` data (e.g. matrix input IRN/JCN, IRN_loc/JCN_loc, internal graph and frontal matrices) mainly rely on matrix indices. All MPI communication will also rely on 32-bit integers. This can lead to significant memory and performance gains with respect to a full 64-bit integer MUMPS version. A full 64-bit integer version can be obtained compiling `MUMPS` with C preprocessing flag -DINTSIZE64 and Fortran compiler option -i8, -fdefault-integer-8 or something equivalent depending on your compiler, and compiling all libraries including MPI, BLACS, ScaLAPACK, LAPACK and BLAS also with 64-bit integers. We refer the reader to the "INSTALL" file provided with the package for details and explanations of the compilation flags controlling integer sizes.

2.5 Upgrading from `MUMPS 4.10.0` to `MUMPS 5.0.2`

2.5.1 Interface with the Metis and ParMetis orderings

Since the release of `MUMPS 4.10.0`, the Metis API has changed. `MUMPS 5.1.2` now assumes that Metis $\geq$ 5.1.0 or ParMetis $\geq$ 4.0.3 are installed, and that the newer versions of Metis/ParMetis are backward compatible with Metis 5.1.0/ParMetis 4.0.3.

It is however possible to continue using `Metis` versions $\leq$ 4.0.3 by forcing the compilation flag -Dmetis4, and to continue using `ParMetis` versions $\leq$ 3.2.0 by forcing the compilation flag -Dparmetis3.

Note that Metis 5.0.1/5.0.2/5.0.3 and ParMetis 4.0.1/4.0.2 have never been supported in `MUMPS`.

2.5.2 Interface with the SCOTCH and PT-SCOTCH orderings

`MUMPS 4.10.0` was not compatible with `SCOTCH 6.x`. `MUMPS 5.1.2` is compatible with both `SCOTCH 5.1.x` and `SCOTCH 6.0.x` and we recommend using the latest version of `SCOTCH`[1].

Since SCOTCH version 6.0.0, the PT-SCOTCH library does not include the SCOTCH library. So during the link phase, the SCOTCH library must be provided to MUMPS. It can be easily done by adding "-lscotch" to the LSCOTCH variable in your Makefile.inc file.

Unfortunately, there is a problem in the SCOTCH 6.0.0 package which is making it unusable with MUMPS. You should update your version of SCOTCH to 6.0.1 or later.

2.5.3 ICNTL(10): iterative refinement

In both `MUMPS 4.10.0` and `MUMPS 5.1.2`, `ICNTL(10)` indicates the maximum number of iterative refinement steps during the solve phase, with a default value of 0 meaning no iterative refinement.

In `MUMPS 4.10.0`, if the user sets `ICNTL(10)` to a negative value, then the value was treated as 0, no iterative refinement. This is not the case with `MUMPS 5.1.2`, where negative values are interpreted differently (fixed number of iterative refinement steps).

This modification should not affect normal users, since it was not natural to reset `ICNTL(10)` to a negative value (instead of the default value of 0) in order to forbid iterative refinement in `MUMPS 4.10.0`.

2.5.4 ICNTL(11): error analysis

Whereas all positive values of `ICNTL(11)` were producing the same statistics in `MUMPS 4.10.0`, `ICNTL(11)`=1, `ICNTL(11)`=2 and `ICNTL(11)` $>$ 2 now have a different behaviour in `MUMPS 5.1.2`, as shown in Table 1.

---

[1]See http://gforge.inria.fr/projects/scotch/

ICNTL(11) value	MUMPS 4.10.0 meaning	MUMPS 5.1.2 meaning
< 0	No error analysis	No error analysis
0	No error analysis (default)	No error analysis(default)
1	Full statistics	Full statistics
2	Full statistics	Main statistics (recommended)
> 2	Full statistics	Defaults to 0, no error analysis

Table 1: Backward compatibility issues between MUMPS 4.10.0 and MUMPS 5.1.2 for ICNTL(11). Full statistics include condition numbers estimates and forward error estimate, which are very expensive to compute.

The main reason for this change is that, because backward error analysis already provides a good indication of the quality of the computed solution, most users might not want to compute forward error analysis and condition numbers estimates in all cases.

2.5.5 ICNTL(20): sparse right-hand sides

The range of values for ICNTL(20) has been extended to better control an internal feature that exploits sparsity of the right-hand sides during the solution phase. This extension should be transparent in practice, since the authorized values for ICNTL(20) were 0 (dense right-hand side expected) and 1 (sparse right-hand sides expected) in MUMPS 4.10.0, which will be correctly interpreted in MUMPS 5.1.2.

One minor difference is in the interpretation of ICNTL(20)=2 or 3 in MUMPS 4.10.0. Whereas both values (2 and 3) are out-of-range values in MUMPS 4.10.0 and treated as 0 (dense right-hand side expected), they now also mean in MUMPS 5.1.2 that the right-hand sides should be provided in sparse form.

Another difference is in the treatment of duplicate entries in the sparse right-hand sides. In MUMPS 4.10.0 the last entry is used whereas in MUMPS 5.1.2 duplicate entries are summed.

We refer the reader to the description of ICNTL(20) for a precise description of sparse right-hand sides and an explanation of the differences between values 1, 2, and 3.

2.5.6 ICNTL(4): Control of the level of printing

By default, some printings that were appearing in MUMPS 4.10.0 no longer appear in MUMPS 5.1.2. This is because, when ICNTL(4)< 2, some diagnostic messages were printed in MUMPS 4.10.0 whereas they should not have been printed. This change of behaviour should thus be considered as a bug correction between MUMPS 4.10.0 and the new version, rather than a backward compatibility issue.

In order to have such messages printed as in MUMPS 4.10.0 with the latest version, please set the value of ICNTL(4) to 2. Please also refer to the detailed descriptions of ICNTL(4), ICNTL(1), ICNTL(2), and ICNTL(3).

2.5.7 License

The license for public versions of MUMPS is CeCILL-C. See also Section 12.

3 Main functionalities of MUMPS 5.1.2

We describe here the main functionalities of the MUMPS solver. These functionalities are controlled by the components of the arrays mumps_par%ICNTL and mumps_par%CNTL. The user should refer to Section 5 and Section 6 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 4). For the sake of clarity and when no confusion is possible, we refer to them only by their component name. For example, we use ICNTL to refer to mumps_par%ICNTL.

3.1 Input matrix format

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

The input matrix can be supplied as *assembled* in coordinate format either on a single processor or distributed over the processors. Otherwise, it can be supplied in *elemental format*, but in this case it can be input only centrally on the host.

For full details on how these formats are handled by `MUMPS`, see Subsection 5.2.2.1 and Subsection 5.2.2.2, respectively for the assembled centralized and assembled distributed formats, and see Subsection 5.2.2.3 for the elemental format .

By default, the input matrix is assumed to be provided in assembled format and centralised on the host processor.

3.2 Preprocessing

During the analysis phase, it is possible to preprocess the matrix to make easier/cheaper the numerical factorization. The package offers a range of symmetric orderings to preserve sparsity, but also other preprocessing 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 :

$$\mathbf{A_{pre}} \;=\; \mathbf{P}\,\mathbf{D_r}\;\mathbf{A}\;\mathbf{Q_c}\,\mathbf{D_c}\,\mathbf{P^T}\,, \tag{5}$$

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 preserve 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. This option is controlled by `ICNTL(28)`. The sequential computation is controlled by `ICNTL(7)` whereas the parallel computation is controlled by `ICNTL(29)`.

  − In the case where the symmetric permutation is computed sequentially, an important range of ordering options is offered including the approximate minimum degree ordering (AMD, [8]), an approximate minimum degree ordering with automatic quasi-dense row detection (QAMD, [2]), 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, [43]), the SCOTCH package [39] from the University of Bordeaux 1, and the Metis package from Univ. of Minnesota [34]. A user-supplied permutation can also be provided and the pivot order must be set by the user on the host in the array `PERM_IN` (see Subsection 5.4.2).

  − When the symmetric permutation is computed in parallel, possible orderings are computed by PT-SCOTCH or ParMetis, which must have been installed by the user.

- 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 [25, 26]. For symmetric matrices this permutation can also be used to constrain the symmetric permutation $\mathbf{P}$ (see `ICNTL(12)`). 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)` (see next paragraph).

- 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 either during the analysis phase or at the beginning of the factorization phase.

Furthermore, preprocessing strategies for symmetric indefinite matrices, as described in [27], can be applied and also lead to scaling arrays; they are controlled by `ICNTL(12)`.

3.3 Post-processing facilities

3.3.1 Iterative refinement

A well known and simple technique to improve the accuracy of the solution of linear systems is the use of *iterative refinement*. It consists in refining an initial solution obtained after solution phase as described in Algorithm 1.

repeat
    Solve $A\Delta x = r$ using the computed factorization
    $x = x + \Delta x$
    $r = b - Ax$
    The computed backward error ω (see Subsection 3.3.2)
until $\omega < \alpha$

Algorithm 1: Iterative refinement. At each step, backward errors are computed and compared to α, the *stopping criterion.*

It has been shown [21] that with only two to three steps of iterative refinement the solution can often be significantly improved. For this reason, alternatively to Algorithm 1, a simple variant of iterative refinement can be used with a fixed number of steps and thus without convergence test (see Subsection 5.5).

The use of iterative refinement can be particularly useful if static pivoting has been used during factorization (see Subsection 3.9).

In MUMPS, iterative refinement can be optionally performed after the solution step using the parameter ICNTL(10).

3.3.2 Error analysis and statistics

MUMPS enables the user to perform classical error analysis based on the residuals. We calculate an estimate of the sparse backward error using the theory and metrics developed in [21]. We use the notation $\bar{\mathbf{x}}$ for the computed solution and a modulus sign on a vector or a matrix to indicate the vector or matrix obtained by replacing all entries by their moduli. The scaled residual

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

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

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

is computed instead, where $\mathbf{A}_i$ is row i of $\mathbf{A}$. In [21], the largest scaled residual in Equation (6), is defined as ω_1 and the largest scaled residual in Equation (7) as ω_2. If all equations are in category (1), ω_2 is zero. ω_1 and ω_2 are the two backward errors.

Then, 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

$$|\delta\mathbf{A}_{ij}| \leq \max(\omega_1, \omega_2)|\mathbf{A}|_{ij},$$

and $|\delta\mathbf{b}_i| \leq \max(\omega_1|\mathbf{b}|_i, \omega_2\|\mathbf{A}_i\|_\infty \|\bar{\mathbf{x}}\|_\infty)$. 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 $\mathbf{A}_{ij}=0$.

Finally, if the user can afford a significantly more costly error analysis, condition numbers $cond_1$ and $cond_2$ for the linear system (not just the matrix) can also be returned together with an upper bound of the forward error of the computed solution :

$$\frac{\|\delta \mathbf{x}\|_\infty}{\|\mathbf{x}\|_\infty} \leq \omega_1 \ cond_1 + \omega_2 \ cond_2$$

This option is controlled by `ICNTL(11)`.

3.4 Null pivot detection

`MUMPS` gives the possibility to detect null pivots of a matrix during factorization. This option is controlled by `ICNTL(24)`. The number of null pivots provides an estimate of the rank deficiency.

At the solution phase, one of the possible solution of the deficient system $AX = B$ can be computed using the control parameter `ICNTL(25)` (Subsection 3.5).

It is also possible to compute all or a part of the null vectors (that is the vectors solving $AX = 0$) associated to these null pivots using the same control parameter `ICNTL(25)`.

3.5 Computation of a solution of a deficient matrix and of a null space basis

Using the `ICNTL(25)` parameter, `MUMPS` gives the possibility to compute one of the possible solutions of $AX = B$ of a matrix that has been found deficient during factorization if a zero-pivot detection option was requested (Subsection 3.4).

The same control parameter (`ICNTL(25)`) can be used to compute a part or the complete null space basis (that is $AX = 0$) of the matrix that has been found deficient during factorization.

3.6 Solving the transposed system

Given a sparse matrix $\mathbf{A}$, the system $\mathbf{AX} = \mathbf{B}$ or $\mathbf{A^T X} = \mathbf{B}$ can be solved during the solve phase, where $\mathbf{A}$ is a square matrix of order n and $\mathbf{X}$ and $\mathbf{B}$ are matrices of order n by $nrhs$. This is controlled by `ICNTL(9)`. Note that in the case where the forward elimination (see Subsection 3.7) is performed during the factorization, solving the transposed system is not allowed.

3.7 Forward elimination during factorization

It is possible to perform the forward elimination (Subsection 5.12) of the right-hand sides (Equation (3)) during the factorization.

If the forward elimination is performed during the factorization, some factors may be discarded (Subsection 5.10) because in this case only the backward substitution (Equation (4)) needs to be performed during the solution phase. This option makes much sense in an out-of-core context (Subsection 3.13), where the solution phase involves loading factors from disk during both the forward (Equation (3)) and backward (Equation (4)) substitutions and can be particularly costly, or when the factors have to be used only once. Note that in the case where the forward elimination is performed during the factorization, solving the transposed system is not allowed.

3.8 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 file named "INSTALL" at the root of the `MUMPS` distribution.

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`,

3.9 Numerical pivoting

The goal of pivoting is to ensure a good numerical accuracy during Gaussian elimination. A widely used technique is known as *partial pivoting*. Considering an unsymmetric matrix, at step i of the factorization we first determine k such that $|a_{k,i}| = \max_{l=i:n} |a_{l,i}|$. Rows i and k are swapped in $\mathbf{A}$ (and the permutation information is stored in order to apply it to the right-hand side $\mathbf{b}$) before dividing the column by the pivot and performing the rank-one update. The advantage of this approach is that it bounds the growth factor and improves the numerical stability.

Unfortunately, in the case of sparse matrices, numerical pivoting prevents a full static prediction of the structure of the factors: it dynamically modifies the structure of the factors, thus forcing the use of dynamic data structures. Numerical pivoting can thus have a significant impact on the fill-in and on the amount of floating-point operations. To limit the amount of numerical pivoting, and stick better to the sparsity predictions done during the symbolic factorization, partial pivoting can be relaxed, leading to the *partial threshold pivoting* strategy:

In the *partial threshold pivoting* strategy, a pivot $a_{i,i}$ is accepted if it satisfies:

$$|a_{i,i}| \geq u \times \max_{k=i:n} |a_{k,i}|, \tag{8}$$

for a given value of u, $0 \leq u \leq 1$. This ensures a growth factor limited to $1 + 1/u$ for the corresponding step of Gaussian elimination. In practice, one often chooses $u = 0.1$ or $u = 0.01$ as a default threshold and this generally leads to a stable factorization. The threshold u can be set using `CNTL(1)`.

It is possible to perform the pivot search on the row rather than on the column with similar stability.

In the multifrontal method, once a frontal matrix is formed, we cannot choose a pivot outside the fully-summed block, because the corresponding rows are not fully-summed. Once all possible pivots in the block of candidate pivots have been eliminated, if no other pivot satisfies the partial pivoting threshold, some rows and columns remain unfactored in the front. Those are then delayed to the frontal matrix of the parent, as part of the contribution block (*delayed pivots*). Note that because of the delayed pivots fill-in the parent node will occur.

The same type of approach is applied to the symmetric case, but with the constrain that we want to maintain the symmetry of the frontal matrices.

In order to avoid the complications due to numerical pivoting, perturbation techniques can be applied (*static pivoting*): a pivot smaller than a threshold in absolute value is replaced by this threshold. In this case it is recommended do use iterative refinement (see Subsection 3.3.1) to improve the approximate solution.

In MUMPS, static pivoting (`CNTL(4)`) and numerical pivoting (`CNTL(1)`) are combined at runtime.

A comparison of approaches based on static pivoting with approaches based on numerical pivoting in the context of high-performance distributed solvers can be found in [14].

3.10 The working host processor

The host processor is the one with rank 0 in the communicator provided to MUMPS. By setting the variable PAR to 1 (see Subsection 5.1.3), 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.

3.11 MPI-free version

It is possible to use MUMPS with a single MPI process by limiting the number of processors to one, or by passing to the solver a communicator consisting of a single MPI process. However, the link phase still requires the MPI, BLACS, and ScaLAPACK libraries in this case.

An MPI-free 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 MPI. Note that in this case parallel ordering packages such as ParMetis or PT-SCOTCH must be disabled during installation. Details on how to build a purely sequential version of MUMPS are available in the INSTALL file available in the MUMPS distribution.

Remark that for the sequential version, the component PAR must be set to 1 (see Subsection 5.1.3). Furthermore, the calling program should not make use of MPI: if the calling program is a parallel MPI code which requires sequential MUMPS, a parallel version of MUMPS must then be installed, to which a communicator consisting of a single process should be provided. Finally, the MPI-free version can make use of several cores by relying on multithreading (see Section 3.12).

3.12   Combining MPI and multithreaded parallelism

MUMPS supports shared memory, multithreaded parallelism mostly through the use of multithreaded BLAS libraries such MKL, ACML or OpenBLAS. Parts of the MUMPS code, other than BLAS operations, have been parallelized through the use of OpenMP directives which can be activated by adding the appropriate OpenMP compiler and linker option in the OPTF and OPTL variables of the MUMPS Makefile.inc file; this option is, for example, `-fopenmp` for GNU gfortran and `-openmp` for Intel ifort but equivalent options exist for other commonly used compilers. The number of threads within the OpenMP parallel regions of MUMPS can be set through the OMP_NUM_THREADS environment variable. Note that, in most cases, the OMP_NUM_THREADS environment variable conveniently controls both the number of threads in the BLAS library and the MUMPS OpenMP parallel regions.

Modern high performance parallel computers are commonly made of multiple nodes, each equipped with multiple processors and cores; in order to make the best out of their performance, we strongly recommend to use both MPI and multithreading (both parallel BLAS and the MUMPS OpenMP directives). A typical setting uses one MPI process per socket each with as many threads as the number of cores on the socket.

3.13   Out-of-core facility

An out-of-core (disk is used as an extension to main memory) facility is available in both sequential and parallel environments. This option is controlled by ICNTL(22).

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. To significantly reduce the cost of the solve phase in an out-of-core context, we advise performing the forward elimination (see Equation (3)) during the factorization, if this is compatible with your usage of MUMPS (limited amount of dense right-hand sides, no iterative refinement or error analysis (see Subsection 3.7)).

3.14   Determinant

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

Let n be the order of the matrix A, if $A = LU$ (unsymmetric matrices), then

$$det(A) = det(L) \times det(U) = \prod_{i=1}^{n} U_{ii}$$

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 computed when requested by the user. If the user is only interested in the determinant, he/she may tell MUMPS that the factor matrices can be discarded (see Subsection 5.10), significantly reducing the storage requirements.

3.15 Computing selected 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 jth 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 jth column of the inverse are requested [45, 41, 15]. If an LU factorization of A had been computed, a_{ij}^{-1}, the (i, j) entry of A^{-1}, is obtained by solving successively the two triangular systems:

$$y = L^{-1}e_j \tag{9}$$

$$a_{ij}^{-1} = (U^{-1}y)_i \tag{10}$$

MUMPS provides a functionality, controlled by ICNTL(30), to compute a set of entries of $\mathbf{A}^{-1}$, while avoiding most of the computations on explicit zeros in Equations (9) and (10). The list of entries of $\mathbf{A}^{-1}$ to be computed and the memory for those entries should be provided as a sparse right-hand side (see Subsection 5.13.2). In a parallel environment it is not so natural to combine parallelism and exploiting sparsity. Recent work based on [41, 16] to exploit parallelism is provided in this release.

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

MUMPS provides the possibility to perform a partial factorization of the complete input matrix and to return the corresponding Schur matrix, that is the part of the matrix that has been updated but that is still to be factorized. This option is controlled by ICNTL(19).

Let us consider a partitioned matrix (here with an unsymmetric matrix) where the variables of $\mathbf{A}_{2,2}$ 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 all the variables belonging to the Schur last.

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_{1,1} & 0 \\ \mathbf{L}_{2,1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{U}_{1,1} & \mathbf{U}_{1,2} \\ 0 & \mathbf{S} \end{pmatrix} \tag{11}$$

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

The user must specify on entry to the analysis phase the list of indices of the Schur matrix, corresponding to the variables of $\mathbf{A}_{2,2}$. MUMPS returns to the user, on exit of the factorization phase, the Schur complement matrix $\mathbf{S}$, as a full matrix but with different type of distribution (see Subsection 5.14 for more details.)

This partial factorization can be used to solve $\mathbf{Ax} = \mathbf{b}$ in different ways using the ICNTL(26) parameter. It can be used to solve the linear system associated with the "interior" variables or to handle a reduced/condensed right-hand-side as described in the following discussion.

- *Compute a partial solution (ICNTL(26) = 0)*:
  The solve is performed on the internal problem:

  $$\mathbf{A}_{1,1}\mathbf{x}_1 = \mathbf{b}_1.$$

  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*:

$$\begin{pmatrix} \mathbf{L}_{1,1} & 0 \\ \mathbf{L}_{2,1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{U}_{1,1} & \mathbf{U}_{1,2} \\ 0 & \mathbf{S} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \qquad (12)$$

First solving:

$$\begin{pmatrix} \mathbf{L}_{1,1} & 0 \\ \mathbf{L}_{2,1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \qquad (13)$$

And thereafter:

$$\begin{pmatrix} \mathbf{U}_{1,1} & \mathbf{U}_{1,2} \\ 0 & \mathbf{S} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \qquad (14)$$

1. *Reduction/condensation phase (`ICNTL(26) = 1`)*:
   The Equation (13) can be solved setting `ICNTL(26)`=1, and the intermediate y vector will be computed, but only the y_2 vector is returned to the user. Note that y_2 is often referred to as the reduced/condensed right-hand-side.

2. *Using the Schur matrix*:
   Thenafter the solution $Sx_2 = y_2$ has to be computed. It is the responsibility of the user to compute x_2 using the Schur matrix S given on output of the factorization phase.

3. *Expansion phase (`ICNTL(26) = 2`)*:
   Given x_2 and y_1, it is possible to compute x_1, such that $U_{11}x_1 = y_1 - U_{12}x_2$, using the option `ICNTL(26)`=2 . Note that the package uses y_1 computed (and stored in the main MUMPS structure) during the first step (`ICNTL(26)`=1) and provides the complete solution x 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.

3.17 Block Low Rank (BLR) multifrontal factorization

A preliminary low-rank feature that allows decreasing the complexity of sparse direct solvers on problems arising from partial differential equations is provided for experimentation purpose.

Matrices coming from elliptic Partial Differential Equations (PDEs) have been shown to have a low-rank property: well defined off-diagonal blocks of their Schur complements can be approximated by low-rank products. Given a suitable ordering of the matrix which gives to the blocks a geometrical meaning, such approximations can be computed using an Singular-Value Decomposition (SVD) or a Rank-Revealing QR factorization (RRQR). The resulting representation offers a substantial reduction of the memory requirement and gives efficient ways to perform many of the basic dense algebra operations. Several strategies have been proposed in the literature to exploit this property.

We have designed a so-called Block Low Rank (BLR) format (see [3]) to reduce the memory footprint and the computational complexity of our multifrontal solver. The compression of BLR blocks is based on a truncated RRQR algorithm.

At each node of the multifrontal computational graph, a partial factorization (part corresponding to the fully summed variables, see Figure 1(a)) of a dense frontal matrix is performed. The so called contribution block (CB in the figure) will correspond to the local Schur matrix built at the end of the partial factorization of the front. As illustrated in Figure 1 the BLR factorization is performed by panels of size the size of a BLR block. On the left-hand side (a) of the figure, a normal full-rank (FR) factorization of the panel is first performed leading to dense FR L and U factor blocks (colored in (a)). The block structure follows the flat block low rank (BLR) structure of the front. Truncated Rank-Revealing QR factorization (RRQR), controlled by a numerical threshold (referred to as ε in Subsection 5.15), is then performed on each off-diagonal block (as indicated in Figure 1(b)). Blocks in low rank form are then used

to update the right looking part of the frontal matrix (grey in (b)) at a lower cost than a full-rank normal update. Finally as a temporary solution (since we do not have yet a BLR solution phase) each low-rank block is decompressed back into the front. This variant is referred to as *FSCU* because of the order in which operations are performed (Factorization, Solve, Compress, Update).

(a) Full-Rank (FR) panel factorization (b) Low rank panel compression and update
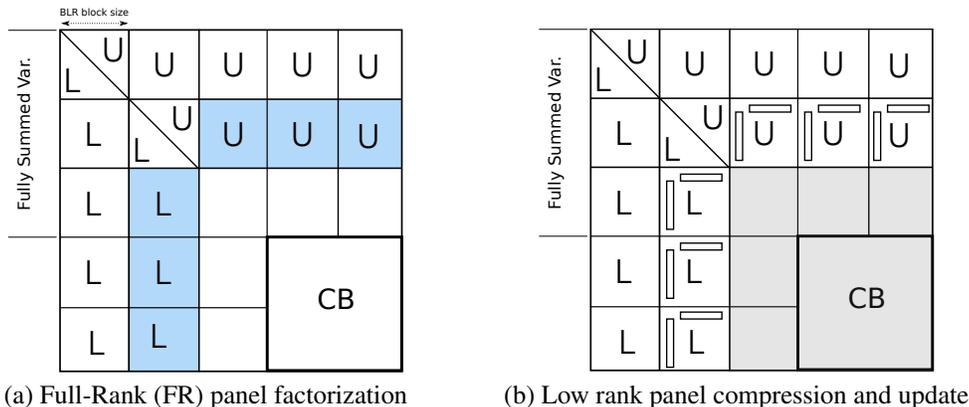
Figure 1: Block low rank factorization of one panel of a the partial factorization of a frontal matrix.

With this feature MUMPS can be used to provide an approximation of the solution controlled by a numerical threshold or an approximate factorization that can be used as a preconditioner (see [47]). The theoretical analysis in [5] proves that the BLR method can reduce the asymptotic complexity of the multifrontal factorization. The work presented in [5] also introduces novel variants that can further reduce the complexity of the BLR factorization by using different strategies to perform the updates in the frontal matrices and to handle numerical pivoting. The theoretical analysis therein shows that the most advanced BLR variants can achieve a complexity which is comparable to that obtained with hierarchical formats such as $\mathcal{H}-$matrices. These novel variants are not yet implemented in MUMPS and only the FSCU one explained above is currently available.

We provide a preliminary version of this functionality that is mainly controlled by a numerical parameter defining the accuracy of the low rank approximations (see Subsection 5.15). We have observed in practice that the backward error of the final solution is closely related to this numerical parameter. In this preliminary version, the memory compression is not yet effective, but large gains can already be obtained in terms of factorization time and operation count [7, 6, 4].

4 User interface and available routines

In the Fortran 95 interface (see Section 9 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 95 derived datatype [SDCZ]MUMPS_STRUC defined in [sdcz]mumps_struc.h.[2]

The interface is the same for the MPI-free version (see Subsection 3.11), only the compilation process and libraries need be changed. In the case of the parallel version, MPI must be initialized by the user before the first call to [SDCZ]MUMPS is made.

The calling sequence for the DOUBLE PRECISION version may look as follows:

```
INCLUDE 'mpif.h'
INCLUDE 'dmumps_struc.h'
...
INTEGER :: IERR
TYPE (DMUMPS_STRUC) :: mumps_par
```

---

[2]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_struc.h to smumps_struc.h, dmumps_struc.h, cmumps_struc.h or zmumps_struc.h.

```
        . . .
        CALL MPI_INIT(IERR)
        . . .
        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)
```

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

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

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

Components of the structure `[SDCZ]MUMPS_STRUC` that are of interest to the user are shown in Figure 2.

In the following, **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.

```fortran
      INCLUDE '[sdcz]mumps_root.h'
      TYPE [SDCZ]MUMPS_STRUC
        SEQUENCE
C INPUT PARAMETERS
C ----------------
C    Problem definition
C    ------------------
C    Solver (SYM=0 Unsymmetric, SYM=1 Sym. Positive Definite, SYM=2 General Symmetric)
C    Type of parallelism (PAR=1 host working, PAR=0 host not working)
      INTEGER SYM, PAR, JOB
C    Control parameters
C    ------------------
      INTEGER ICNTL(40)
      real CNTL(15)
      INTEGER N   ! Order of input matrix
C    Assembled input matrix : User interface
C    ---------------------------------------
      INTEGER      :: NZ ! Standard integer input + bwd. compat.
      INTEGER(8) :: NNZ ! 64-bit integer input
      real/complex, DIMENSION(:), POINTER :: A
      INTEGER, DIMENSION(:), POINTER :: IRN, JCN
C       Case of distributed matrix entry
C       --------------------------------
      INTEGER      :: NZ_loc  ! Standard integer input + bwd. compat.
      INTEGER(8) :: NNZ_loc ! 64-bit integer input
      INTEGER, DIMENSION(:), POINTER :: IRN_loc, JCN_loc

      real/complex, DIMENSION(:), POINTER :: A_loc
C    Unassembled input matrix: User interface
C    ----------------------------------------
      INTEGER NELT
      INTEGER, DIMENSION(:), POINTER :: ELTPTR, ELTVAR
      real/complex, DIMENSION(:), POINTER :: A_ELT
C    MPI Communicator and identifier
C    -------------------------------
      INTEGER COMM, MYID
C    Ordering and scaling, if given by user (optional)
C    -------------------------------------------------
      INTEGER, DIMENSION(:), POINTER :: PERM_IN
      real, DIMENSION(:), POINTER :: COLSCA, ROWSCA
C INPUT/OUTPUT data  : right-hand side and solution
C -----------------
      real/complex, DIMENSION(:), POINTER :: RHS, REDRHS
      real/complex, DIMENSION(:), POINTER :: RHS_SPARSE
      INTEGER, DIMENSION(:), POINTER :: IRHS_SPARSE, IRHS_PTR
      INTEGER NRHS, LRHS, NZ_RHS, LSOL_loc, LREDRHS
      real/complex, DIMENSION(:), POINTER :: SOL_loc
      INTEGER, DIMENSION(:), POINTER :: ISOL_loc
C OUTPUT data and Statistics
C --------------------------
      INTEGER, DIMENSION(:), POINTER :: SYM_PERM, UNS_PERM
      INTEGER INFO(40)
      INTEGER INFOG(40)   ! Global information (host only)
      real RINFO(20)
      real RINFOG(20) ! Global information (host only)
C    Schur
      INTEGER SIZE_SCHUR, NPROW, NPCOL, MBLOCK, NBLOCK
      INTEGER SCHUR_MLOC, SCHUR_NLOC, SCHUR_LLD
      INTEGER, DIMENSION(:), POINTER :: LISTVAR_SCHUR
      real/complex, DIMENSION(:), POINTER :: SCHUR
C    Mapping if provided by MUMPS
      INTEGER, DIMENSION(:), POINTER :: MAPPING
C    Version number
      CHARACTER(LEN=46) VERSION_NUMBER
C    Name of file to dump a problem in matrix market format
      CHARACTER(LEN=255) WRITE_PROBLEM
C    Out-of-core}
      CHARACTER(LEN=63)  :: OOC_PREFIX
      CHARACTER(LEN=255) :: OOC_TMPDIR
      END TYPE [SDCZ]MUMPS_STRUC
```

Figure 2: Main components of the structure `[SDCZ]MUMPS_STRUC` defined in `[sdcz]mumps_struc.h`.

5 Application Program Interface

In this section, we describe the main components of the variable mumps_par of datatype
`[SDCZ]MUMPS_STRUC` (see Section 4), that must be set by the user, or that are returned to the user.
The parameters controlling the activation of the main functionalities of the package are provided on input
in the arrays `ICNTL` and `CNTL`. In each section, we describe in detail the entry of the `ICNTL` and `CNTL`
related to the section. The entire list is given in Section 6. Statistics and various information parameters on
output to the solver are then described in Section 7. Information parameters returned by the package might
correspond to values local to each processor (mumps_par%RINFO and mumps_par%INFO) or might be
global information and available on all processors (mumps_par%RINFOG and mumps_par%INFOG).

For the sake of clarity (and when no confusion is possible), we refer to components of the structure
only by their component name; for example, we use `ICNTL` to refer to `mumps_par%ICNTL`.

5.1 General

5.1.1 Initialization, Analysis, Factorization, Solve, Termination (`JOB`)

JOB (integer) must be initialized by the user on all processors before a call to `MUMPS`. It controls the
main actions taken by `MUMPS`. It is not altered by `MUMPS`. Possible values of JOB are:

–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
`[SDCZ]MUMPS_STRUC` (such as ICNTL), 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

- `COMM`,
- `SYM`, and
- `PAR`.

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

  Furthermore, after a call with JOB $= -1$, the internal component **MYID** contains the rank of the
  calling processor in the communicator provided to `MUMPS`. Thus, the test "(`MYID == 0`)" may be
  used to identify the host processor (see Subsection 3.10).

  Finally, the version number is returned in `VERSION_NUMBER` (see Subsection 5.1.2).

–2 terminates an instance of the package. All data structures associated in `mumps_par` with the instance,
except those provided by the user, are deallocated. It should be called by the user only when no
further calls to `MUMPS` with this instance are required. In order to avoid memory leaks, it must also
be called before a further `JOB` $= -1$ call with the same argument `mumps_par`. When out-of-core
was used (`ICNTL(22)` =1), the out-of-core factor files are deleted.

1 performs the analysis. In this phase, `MUMPS` chooses pivots from the diagonal using a selection criterion
to preserve sparsity, using the pattern of the matrix **A** input by the user. Several formats and
distributions onto the processors are available to input matrix (see Subsection 5.2.2). 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 Subsection 5.4) in
  which case only the necessary information for a `JOB`=2 call will be generated.

  If a preprocessing based on the numerical values is requested (see Subsection 5.3 and `ICNTL(6)`),
  then the numerical values of the original matrix **A** must also be provided by the user during the
  analysis phase, and scaling vectors are optionally computed.

  Note that a call to `MUMPS` with `JOB`=1 must be preceded by a call with `JOB` $= -1$ on the same
  instance.

2 performs the factorization. It uses the numerical values of the matrix **A** provided by the user (see
Subsection 5.2.2) and the information from the analysis phase (`JOB`=1) to factorize the matrix **A**.

  The actual pivot sequence used during the factorization may slightly differ from the sequence
  returned by the analysis (see Subsection 5.4.1) if the matrix **A** is not diagonally dominant.

An option exists for the user to input scaling vectors or let MUMPS compute automatically such vectors (see Subsection 5.3.2 and ICNTL(8)) just before the numerical factorization.

A call to MUMPS with JOB=2 must be preceded by a call with JOB=1 on the same instance.

3 computes the solution. 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). Several possibilities are given to input the right-hand side matrix **B** and to ouput the solution matrix **X**. The structure component mumps_par%RHS must be set by the user (on the host only) before a call with JOB=3 (see Subsection 5.13). This solution phase can also be used to compute the null space basis of singular matrices (see ICNTL(25)), provided that "null pivot" detection (ICNTL(24)) was on and that the deficiency obtained INFOG(28) was different from 0.

A call to MUMPS with JOB=3 must be preceded by a call with JOB=2 (or JOB=4) on the same instance.

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.

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.

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.

5.1.2 Version number

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 9 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 "5.1.2". 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).

5.1.3 Control of parallelism (COMM, PAR)

COMM (integer) must be set by the user on all processors before the initialization phase (JOB $= -1$) and must not be changed in further calls. It must be set to a valid MPI communicator that will be used for message passing inside MUMPS. It is not altered by MUMPS and a copy of the communicator is kept internally by the package until a call to the termination phase (JOB $= -2$). 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.

PAR (integer) must be initialized by the user on all processors before the initialization phase (JOB $= -1$) and is accessed by MUMPS only during this phase. It is not altered by MUMPS and its value is communicated internally to the other phases as required. Possible values for PAR are:

0 : The host is not involved in the parallel steps of the factorization and solve phases. The host will only hold the initial problem, perform symbolic computations during the analysis phase, distribute data, and collect results from other processors.

1 : The host is also involved in the parallel steps of the factorization and 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 (the host) is used by the package.

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.

5.2 Input Matrix

5.2.1 Matrix type (`SYM`)

The user must set the variable `SYM` to indicate which kind of matrix has to be factorize and consequently, which factorization has to be performed.

SYM (integer) must be initialized by the user on all processors before the initialization phase (`JOB` = −1) and is accessed by `MUMPS` only during this phase. 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 assumed to be symmetric positive definite so that pivots are taken from the diagonal without numerical pivoting during the factorization. With this option, non-positive definite matrices that do not require pivoting can also be treated in certain cases (see remark below).

    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`.

Remark for symmetric matrices (SYM=1). When `SYM`=1 is indicated by the user, an $\mathbf{LDL^T}$ factorization (in opposition to Cholesky factorization which requires positive diagonal pivots) of matrix **A** is performed internally by the package, and numerical pivoting is switched off. Therefore, this setting works for classes of matrices more general than positive definite matrices, including matrices with negative pivots. However, this feature depends on the use of the ScaLAPACK library (see `ICNTL(13)`) to factorize the last dense block in the factorization of **A**. More precisely,

- if ScaLAPACK is allowed for the last dense block (default in parallel, `ICNTL(13)`=0), then the presence of negative pivots in the part of the factorization processed with ScaLAPACK (subroutine P_POTRF) will raise an error and the code -40 is then returned in INFOG(1);

- if ScaLAPACK is not used (`ICNTL(13)` >0, or sequential execution, or last dense block detected to be too small), then negative pivots are allowed and the factorization will work for some classes of non-positive definite matrices where numerical pivoting is not necessary, e.g., symmetric negative matrices.

The successful factorization of a symmetric matrix with `SYM`=1 is thus not an indication that the matrix provided was symmetric positive definite. In order to verify that a matrix is positive definite, the user can check that the number of negative pivots or inertia (`INFOG(12)`) is 0 on exit from the factorization phase. Another approach to suppress numerical pivoting on symmetric matrices which is compatible with the use of ScaLAPACK (see `ICNTL(13)`) consists in setting `SYM`=2 (general symmetric matrices) with the relative threshold for pivoting `CNTL(1)` set to 0 (recommended strategy).

5.2.2 Matrix format

The formats of the input matrix and its distribution onto the processors are controlled by `ICNTL(5)` and `ICNTL(18)`, respectively.

ICNTL(5) controls the matrix input format

    Phase: accessed by the host and only during the analysis phase

    Possible variables/arrays involved: N, NNZ (or NZ for backward compatibility), IRN, JCN, NNZ_loc (or NZ_loc for backward compatibility), IRN_loc, JCN_loc, A_loc, NELT, ELTPTR, ELTVAR, and A_ELT

    Possible values :

    0 : assembled format. The matrix must be input in the structure components N, NNZ (or NZ), IRN, JCN, and A if the matrix is centralized on the host (see Subsection 5.2.2.1) or in the structure components N, NNZ_loc (or NZ_loc)), IRN_loc, JCN_loc, A_loc if the matrix is distributed (see Subsection 5.2.2.2).

1 : elemental format. The matrix must be input in the structure components N, NELT, ELTPTR, ELTVAR, and A_ELT (see Subsection 5.2.2.3).

Any other values will be treated as 0.

Default value: 0 (assembled format)

Related parameters: `ICNTL(18)`

Remarks: NNZ and NNZ_loc are 64-bit integers (NZ and NZ_loc are 32-bit integers kept for backward compatibility and will be obsolete in future releases).

Parallel analysis (`ICNTL(28)` =2) is only available for matrices in assembled format and, thus, an error will be raised for elemental matrices (ICNTL(5)=1).

Elemental matrices can be input only centralized on the host (`ICNTL(18)` =0).

ICNTL(18) defines the strategy for the distribution of the input matrix (only for assembled matrix).

Phase: accessed by the host during the analysis phase.

Possible values :

0 : the input matrix is centralized on the host (see Subsection 5.2.2.1).

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 entries distributed according to the mapping on entry to the numerical factorization phase (see Subsection 5.2.2.2).

2 : the user provides the structure of the matrix on the host at analysis, and the distributed matrix entries on all slave processors at factorization. Any distribution is allowed (see Subsection 5.2.2.2).

3 : user directly provides the distributed matrix, pattern and entries, input both for analysis and factorization (see Subsection 5.2.2.2).

Other values are treated as 0.

Default value: 0 (input matrix centralized on the host)

Related parameters: `ICNTL(5)`

Remarks: In case of distributed matrix, we recommand options 2 or 3. Among them, we recommand option 3 which is easier to use. Option 1 is kept for backward compatibility but is deprecated and we plan to suppress it in a future release.

5.2.2.1   Centralized assembled matrix (`ICNTL(5)` =0 and `ICNTL(18)` =0).   In the following, an example of an unsymmetric 3x3 assembled matrix with 5 nonzeros is given.

$$\mathbf{A} = \begin{pmatrix} a_{11} & & \\ & a_{22} & a_{23} \\ a_{31} & & a_{33} \end{pmatrix}$$

Order of the matrix	N	= 3					
Nonzeros in the matrix	NNZ	= 5					
array of row indices	IRN	$[1:NNZ]$	=	2	3	2	1	3
array of col indices	JCN	$[1:NNZ]$	=	3	1	2	1	3
array of values	A	$[1:NNZ]$	=	a_{23}	a_{31}	a_{22}	a_{11}	a_{33}

Note that the elements of the matrix can be input in any order.

The following components of [SDCZ]MUMPS_STRUC hold the matrix in centralized assembled format:

mumps_par%**N** (integer) is the order of the matrix **A**, N > 0. It must be set by the user on the host before analysis. It is not altered by MUMPS.

mumps_par%**NNZ** (integer(8)) is the number of nonzero entries being input, NNZ > 0. It must be set by the user on the host before analysis. (Note that mumps_par%**NZ** (integer) is also available for backward compatibility.) It is not altered by MUMPS.

mumps_par%**IRN** and mumps_par%**JCN** (integer pointer arrays, dimension NNZ) contain the row and column indices, respectively, for the matrix entries. They must be set by the user on the host before analysis. They are not altered by MUMPS.

mumps_par%**A** (**real/complex** pointer array, dimension NNZ) must be set by the user in such a way that A(k) is the value of the entry in row IRN(k) and column JCN(k) of the matrix. It 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). A is not altered by MUMPS. Duplicate entries are summed and all entries with IRN(k) or JCN(k) out-of-range are ignored.

Note that, in the case of symmetric matrices (SYM=1 or 2), only half of the matrix should be provided. For example, only the lower triangular part of the matrix (including the diagonal) or only the upper triangular part of the matrix (including the diagonal) can be provided in IRN, JCN, and A. More precisely, a diagonal nonzero a_{ii} must be provided as A(k)=a_{ii}, IRN(k)=JCN(k)=i, and a pair of off-diagonal nonzeros $a_{ij} = a_{ji}$ must be provided either as A(k)=a_{ij} and IRN(k)=i, JCN(k)=j or vice-versa. Again, out-of-range entries are ignored and duplicate entries are summed. In particular, this means that if both a_{ij} and a_{ji} are provided, they will be summed.

5.2.2.2 Distributed assembled matrix (ICNTL(5)=0 and ICNTL(18)=1,2,3). When the matrix is in assembled format (ICNTL(5)=0), we offer several options to distribute the matrix, defined by the control parameter ICNTL(18).

◇ 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);

◇ 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)

The following components of the structure define the distributed assembled matrix input. They are valid for ICNTL(18)=1,2,3, otherwise the user should refer to Subsection 5.2.2.1 for the centralized assembled matrix input.

The following components of [SDCZ]MUMPS_STRUC hold the matrix in distributed assembled format:

mumps_par%**N** (integer) is the order of the matrix **A**, N > 0. It must be set by the user on the host before analysis. It is not altered by MUMPS.

mumps_par%**NNZ** (integer(8)) is the number of entries being input in the definition of **A**, NNZ > 0. (mumps_par%**NZ** (integer) is also available for backward compatibility.) It must be set by the user on the host before analysis if ICNTL(18) = 1 or 2.

mumps_par%**IRN** and mumps_par%**JCN** (integer pointer array, dimension NNZ) contain the row and column indices, respectively, for the matrix entries. They must be set by the user on the host before analysis if ICNTL(18) = 1, or 2. They can be deallocated by the user just after the analysis.

mumps_par%**NNZ_loc** (integer(8)) 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. (mumps_par%**NZ_loc** (integer) is also available for backward compatibility.)

mumps_par%**IRN_loc** and mumps_par%**JCN_loc** (integer pointer array, dimension NNZ_loc) contain the global[3] 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.

---

[3]If the calling application manages both local and global indices, the global indices must be provided.

mumps_par%**A_loc** (**real/complex** pointer array, dimension NNZ_loc) 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).

mumps_par%**MAPPING** (integer array, dimension NNZ) 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. This parameter and the option `ICNTL(18)` = 1 are kept for backward compatibility with previous versions but are deprecated and will be suppressed in a future release.

We recommend the use of options `ICNTL(18)` = 2 or 3 because they are the most flexible options. Furthermore, these options (2 or 3) are in general as efficient as the more complicated (and deprecated) option `ICNTL(18)` =1. Among those two options, `ICNTL(18)` =3 is the simplest and most natural one to use. `ICNTL(18)` =2 should only be used if the application has a centralized version of the entire matrix already available on the host processor.

Again, out-of-range entries are ignored and duplicate entries are summed. In particular, if an entry a_{ij} is provided both as (IRN_loc(k1), JCN_loc(k1), A_loc(k1)) on a process P1 and as (IRN_loc(k2), JCN_loc(k2), A_loc(k2)) on a process P2, the corresponding numerical value considered for a_{ij} is the sum of A_loc(k1) on P1 and A_loc(k2) on P2. This also means that it is possible to only perform local assemblies inside each MPI process and that entries that are common to several MPI processes (which may typically correspond to interface variables) will be summed internally by the `MUMPS` package without the user having to take care of communications to assemble those entries.

5.2.2.3 Elemental matrix (`ICNTL(5)`=1 and `ICNTL(18)`=0). In the following, an example of elemental matrix with two elements is given.

$$\mathbf{A}_1 = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} \begin{pmatrix} -1 & 2 & 3 \\ 2 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \quad \mathbf{A}_2 = \begin{matrix} 3 \\ 4 \\ 5 \end{matrix} \begin{pmatrix} 2 & -1 & 3 \\ 1 & 2 & -1 \\ 3 & 2 & 1 \end{pmatrix}$$

$$\mathbf{A} = \begin{pmatrix} -1 & 2 & 3 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 1 & 1 & 3 & -1 & 3 \\ 0 & 0 & 1 & 2 & -1 \\ 0 & 0 & 3 & 2 & 1 \end{pmatrix} = \mathbf{A}_1 + \mathbf{A}_2$$

- N=5   NELT=2   NVAR=6 $\mathbf{A} = \sum_{i=1}^{NELT} \mathbf{A}_i$

- ELTPTR   [1:NELT+1]   =   1 4 7
- ELTVAR   [1:NVAR]   =   1 2 3 3 4 5
  A_ELT   [1:NVAL]   =   -1 2 1 2 1 1 3 1 1 2 1 3 -1 2 2 3 -1 1

- Remarks:
  - NVAR = ELTPTR(NELT+1)-1
  - Order of element i: $S_i = ELTPTR(i+1) - ELTPTR(i)$
  - NVAL = $\sum S_i^2$ (unsymmetric) or $\sum S_i(S_i+1)/2$ (symmetric),
  - storage of elements in ELTVAL: by columns

In the current release of the package, a matrix in elemental format must be input centrally on the host (`ICNTL(5)`=1 and `ICNTL(18)`=0). The distributed elemental format is not currently available.

mumps_par%**N** (integer), mumps_par%**NELT** (integer), mumps_par%**ELTPTR** (integer pointer array, dimension NELT+1), mumps_par%**ELTVAR** (integer pointer array, dimension ELTPTR(NELT+1) – 1), and mumps_par%**A_ELT** (**real/complex** pointer array) hold the matrix in elemental format. The following components of the `MUMPS_STRUC` hold the matrix in elemental format:

mumps_par%**N** (integer) is the order of the matrix **A**, N > 0. It is not altered by MUMPS.

mumps_par%**NELT** (integer) is the number of elements being input, NELT > 0. It is not altered by MUMPS.

mumps_par%**ELTPTR** (integer pointer array, dimension NELT+1) is such that 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. ELPTR is not altered by MUMPS.

mumps_par%**ELTVAR** (integer pointer array, dimension ELTPTR(NELT+1) – 1) must be set to the lists of variables of the elements. It is not altered by MUMPS. The variables for element j are stored in positions ELTPTR(j), . . . , ELTPTR(j+1)–1. Out-of-range variables are ignored.

mumps_par%**A_ELT** (**real/complex** pointer array) If N_{p} denotes ELTPTR(p+1)–ELTPTR(p), then the values for element j are stored in positions $K_{\mathrm{j}} + 1, \ldots, K_{\mathrm{j}} + L_{\mathrm{j}}$, where

$\rightarrow$ $K_{\mathrm{j}} = \sum_{\mathrm{p}=1}^{\mathrm{j}-1} N_{\mathrm{p}}{}^2$, and $L_{\mathrm{j}} = N_{\mathrm{j}}{}^2$ in the unsymmetric case (SYM = 0)

$\rightarrow$ $K_{\mathrm{j}} = \sum_{\mathrm{p}=1}^{\mathrm{j}-1} (N_{\mathrm{p}} \cdot (N_{\mathrm{p}} + 1))/2$, and $L_{\mathrm{j}} = (N_{\mathrm{j}} \cdot (N_{\mathrm{j}} + 1))/2$ in the symmetric case (SYM=1 or 2). 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 at the analysis phase (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) and should be passed unchanged when later calling the factorization (JOB=2) and solve (JOB=3) phases. Component A_ELT must be set before the factorization phase (JOB=2).

5.2.3 Writing the input matrix to a file

If the input matrix is in assembled format (centralized or distributed), it is possible to write the matrix at the analysis phase into a file whose name is given in the variable "WRITE_PROBLEM". The "matrix market" format[4] is used to write the matrix.

mumps_par%**WRITE_PROBLEM** (string) must be set by the user before the analysis phase (JOB=1).

If the matrix is distributed, each processor must initialize the variable WRITE_PROBLEM. Each processor will then write its share of the matrix in a file whose name is defined by the string "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.

Furthermore, if a dense right-hand side RHS (see Subsection 5.13.1) is provided on the host before the analysis phase, it is also written in a file with the same name of the matrix file name (WRITE_PROBLEM) but appended by ".rhs".

5.3 Preprocessing: permutation to zero-free diagonal and scaling

The permutation to a zero-free diagonal and the scalings are controlled by ICNTL(6) and ICNTL(8), respectively.

ICNTL(6) computes a permutation to permute the matrix to a zero-free diagonal and/or computes a matrix scaling.

Phase: accessed by the host and only during sequential analysis (ICNTL(28)=1)

Possible variables/arrays involved: optionally UNS_PERM, mumps_par%A, COLSCA and ROWSCA

Possible values :

---

[4]See http://math.nist.gov/MatrixMarket/

0 : No column permutation is computed.

1 : The permuted matrix has as many entries on its diagonal as 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. The numerical values of the original matrix, (mumps_par%A), must be provided by the user during the analysis phase.

3 : Variant of option 2 with different performance. The numerical values of the original matrix (mumps_par%A) must be provided by the user during the analysis phase.

4 : The sum of the diagonal entries of the permuted matrix is maximized. The numerical values of the original matrix (mumps_par%A) must be provided by the user during the analysis phase.

5 : The product of the diagonal entries of the permuted matrix is maximized. Scaling vectors are also computed and stored in COLSCA and ROWSCA, if ICNTL(8) is set to -2 or 77. With these scaling vectors, 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. For unsymmetric matrices, COLSCA and ROWSCA are meaningful on the permuted matrix $\mathbf{A}\ \mathbf{Q_c}$ (see Equation (5)). For symmetric matrices, COLSCA and ROWSCA are meaningful on the original matrix $\mathbf{A}$. The numerical values of the original matrix, mumps_par%A, must be provided by the user during the analysis phase.

6 : Similar to 5 but with a different algorithm. The numerical values of the original matrix, mumps_par%A, must be provided by the user during the analysis phase.

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

Default value: 7 (automatic choice done by the package)

Incompatibility: If the matrix is symmetric positive definite (SYM = 1), or in elemental format (ICNTL(5)=1), or the parallel analysis is requested (ICNTL(28)=2) 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)=1,2,3), then ICNTL(6) is treated as 0.

Related parameters: ICNTL(8), ICNTL(12)

Remarks: *On assembled centralized unsymmetric matrices* (ICNTL(5)=0, ICNTL(18)=0, SYM = 0), if ICNTL(6)=1, 2, 3, 4, 5, 6 a column permutation (based on weighted bipartite matching algorithms described in [25, 26]) is applied to the original matrix to get a zero-free diagonal. The user is advised to set ICNTL(6) to a nonzero value when the matrix is very unsymmetric in structure. On output to the analysis phase, when the column permutation is not the identity, the pointer UNS_PERM (internal data valid until a call to MUMPS with JOB=-2) provides access to the permutation on the host processor (see Subsection 5.3.1). Otherwise, the pointer is not associated. The column permutation is such that entry $a_{i,perm(i)}$ is on the diagonal of the permuted matrix.

On general assembled centralized symmetric matrices (ICNTL(5)=0, ICNTL(18)=0, SYM = 2), if ICNTL(6)=1, 2, 3, 4, 5, 6, the column permutation is internally used to determine a set of recommended 1×1 and 2×2 pivots (see [27] and the description of ICNTL(12) in Subsection 6.1 for more details). 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 UNS_PERM is not associated.

ICNTL(8) describes the scaling strategy

Phase: accessed by the host during analysis phase (that need be sequential ICNTL(28)=1) or on entry to numerical factorization phase

Possible variables/arrays involved: COLSCA,ROWSCA

Possible values :

-2: Scaling computed during analysis (see [25, 26] for the unsymmetric case and [27] for the symmetric case). The user has to provide the numerical values of the original matrix (mumps_par%A) on entry to the analysis.

-1: Scaling provided by the user. Scaling arrays must be provided in COLSCA and ROWSCA on entry to the numerical factorization phase by the user, who is then responsible for allocating and freeing them. If the input matrix is symmetric (SYM= 1 or 2), then the user should ensure that the array ROWSCA is equal to (or points to the same location as) the array COLSCA.

0 : No scaling applied/computed.

1 : Diagonal scaling 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,

7 : Simultaneous row and column iterative scaling based on [42] and [17] computed during the numerical factorization phase.

8 : Similar to 7 but more rigorous and expensive to compute; computed during the numerical factorization phase.

77 : Automatic choice of the value of ICNTL(8) done during analysis.

Default value: 77 (automatic choice done by the package)

Related parameters: ICNTL(6), ICNTL(12)

Remarks: If 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 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 phase. Note that scalings can be efficiently computed during analysis when requested (see ICNTL(6) and ICNTL(12)).

If the input matrix is symmetric (SYM= 1 or 2), then only options –2, –1, 0, 1, 7, 8 and 77 are allowed and other options are treated as 0.

If the input matrix is in elemental format (ICNTL(5) = 1), then only options –1 and 0 are allowed and other options are treated as 0.

If the initial assembled matrix is distributed (ICNTL(18)=1,2,3 and ICNTL(5) = 0), then only options 7, 8 and 77 are allowed, otherwise no scaling is applied.

5.3.1 Permutation to a zero-free diagonal (ICNTL(6))

On assembled centralized unsymmetric matrices (ICNTL(5)=0, ICNTL(18)=0, SYM = 0), if ICNTL(6)=1, 2, 3, 4, 5, 6 a column permutation (based on weighted bipartite matching algorithms described in [25, 26]) is applied to the original matrix to get a zero-free diagonal. The user is advised to set ICNTL(6) to a nonzero value when the matrix is very unsymmetric in structure, or to leave it to its default (automatic) value.

mumps_par%**UNS_PERM** (integer pointer array, dimension N) is returned on the host processor on output to the analysis phase for a centralized unsymmetric matrix, when a column permutation is requested (ICNTL(6) $\neq$0) and if it not the identity. For all other cases, the pointer is not associated. It is allocated internally by MUMPS and provides access to the permutation, and is such that entry $a_{i,perm(i)}$ is on the diagonal of the permuted matrix.

5.3.2 Scaling (ICNTL(6) **or** ICNTL(8))

mumps_par%**COLSCA**, mumps_par%**ROWSCA** (**real** pointer arrays, dimension N) are optional, column and row scaling arrays, respectively, required only by the host. Note that this arrays are **real** also in the complex version.

On input: 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).

On output:

If `ICNTL(6)`=5 or 6, and `ICNTL(8)`=-2 or 77, they are automatically allocated and computed by the package during the analysis phase.

Otherwise, they are automatically allocated and computed by the package during the factorization phase.

5.4 Preprocessing: symmetric permutations

The choice of the symmetric permutation **P**, the so called ordering, from Equation (5) is managed by the control parameters `ICNTL(28)`, `ICNTL(7)` and `ICNTL(29)` defined below:

ICNTL(28) determines whether a sequential or a parallel analysis is performed.

Phase: accessed by the host process during the analysis phase.

Possible values :

- 0: automatic choice.
- 1: sequential computation. In this case the ordering method is set by `ICNTL(7)` and the `ICNTL(29)` parameter is meaningless (choice of the parallel ordering tool).
- 2: parallel computation. A parallel ordering and parallel symbolic factorization is requested by the user. For that, one of the parallel ordering tools (or all) must be available, and the matrix should not be too small. In this case `ICNTL(7)` is meaningless.

Any other values will be treated as 0.

Default value: 0 (automatic choice)

Incompatibility: The parallel analysis is not available when the Schur complement feature is requested (`ICNTL(19)`=1,2 or 3), when a maximum transversal is requested on the input matrix (i.e., `ICNTL(6)`=1, 2, 3, 4, 5 or 6) or when the input matrix is an unassembled matrices (`ICNTL(5)`=1). When the number of processes available for parallel analysis is equal to 1, or when the initial matrix is extremely small, a sequential analysis is indeed performed, even if `ICNTL(28)`=2 (no error is raised in that case).

Related parameters: `ICNTL(7)`, `ICNTL(29)`, `INFOG(32)`

Remarks: Performing the analysis in parallel (`ICNTL(29)`= 2) will enable saving both time and memory. Note that then the quality of the ordering depends on the number of processors used. The number of processors for parallel analysis may be smaller than the number of MPI processes available for MUMPS, in order to satisfy internal constraints of parallel ordering tools. On output, `INFOG(32)` is set to the type of analysis (sequential or parallel) that was effectively chosen internally.

ICNTL(7) computes a symmetric permutation (ordering) to determine the pivot order to be used for the factorization (see Subsection 3.2)

Phase: accessed by the host and only during the *sequential* analysis phase (`ICNTL(28)` = 1).

Possible variables/arrays involved: `PERM_IN`, `SYM_PERM`

Possible values :

- 0 : Approximate Minimum Degree (AMD) [8] 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, ... N) must hold the position of variable i in the pivot order. In other words, row/column i in the original matrix corresponds to row/column `PERM_IN`(i) in the reordered matrix.
- 2 : Approximate Minimum Fill (AMF) is used,
- 3 : SCOTCH[5] [39] package is used if previously installed by the user otherwise treated as 7.
- 4 : PORD[6] [43] is used if previously installed by the user otherwise treated as 7.

---

[5]See http://gforge.inria.fr/projects/scotch/ to obtain a copy.
[6]Distributed within MUMPS by permission of J. Schulze (University of Paderborn).

5 : the Metis[7] [34] package is used if previously installed by the user otherwise treated as 7.

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.

Default value: 7 (automatic choice)

Incompatibility: ICNTL(7) is meaningless if the parallel analysis is chosen (`ICNTL(28)`=2).

Related parameters: `ICNTL(28)`

Remarks: Even when the ordering is provided by the user, the analysis must be performed before numerical factorization.

For *assembled matrices (centralized or distributed)* (`ICNTL(5)`=0) all the options are available.

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.

If the user asks for *a Schur complement matrix* (`ICNTL(19)`= 1, 2, 3) and

- the matrix is <u>assembled</u> (`ICNTL(5)`=0) then only options 0, 1, 5 and 7 are currently available. Other options are treated as 7.

- the matrix is <u>elemental</u> (`ICNTL(5)`=1) only options 0, 1 and 7 are currently available. Other options are treated as 7 which will (currently) be treated as 0 (AMD).

- in both cases (assembled or elemental matrix) if the pivot order is given by the user (ICNTL(7)=1) then the following property should hold: `PERM_IN`(`LISTVAR_SCHUR`(i)) = `N`-`SIZE_SCHUR`+i, for i=1,`SIZE_SCHUR`.

For matrices with *relatively dense rows*, we highly recommend option 6 which may significantly reduce the time for analysis.

On output, the pointer array `SYM_PERM` provides access, on the host processor, to the symmetric permutation that is effectively computed during the analysis phase by the MUMPS package, and `INFOG(7)` to the ordering option that was effectively chosen. In fact, the option corresponding to ICNTL(7) may be forced by MUMPS when for example the ordering option chosen by the user is not compatible with the value of `ICNTL(12)` or the necessary package is not installed.

`SYM_PERM`(i), i=1, ... N, holds the position of variable i in the pivot order. In other words, row/column i in the original matrix corresponds to row/column `SYM_PERM`(i) in the reordered matrix. See also Subsection 5.4.1.

ICNTL(29) defines the parallel ordering tool to be used to compute the fill-in reducing permutation.

Phase: accessed by host process only during the parallel analysis phase (`ICNTL(28)`=2).

Possible variables/arrays involved: `SYM_PERM`

Possible values :

0: automatic choice.

1: PT-SCOTCH is used to reorder the input matrix, if available.

2: ParMetis is used to reorder the input matrix, if available.

Default value: 0 (automatic choice)

Related parameters: `ICNTL(28)`

Remarks: On output, the pointer array `SYM_PERM` provides access, on the host processor, to the symmetric permutation that is effectively considered during the analysis phase, and `INFOG(7)` to the ordering option that was effectively used. `SYM_PERM`(i), (i=1, ... N) holds the position of variable i in the pivot order, see Subsection 5.4.1 for a full description.

---

[7]See `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview` to obtain a copy.

5.4.1 Symmetric permutation vector (`ICNTL(7)` and `ICNTL(29)`)

When the ordering is not provided by the user, the choice of the ordering strategy is controlled by `ICNTL(7)` in case of sequential analysis (`ICNTL(28)`=1), and by `ICNTL(29)` in case of parallel analysis (`ICNTL(28)`=2). In all cases (serial or parallel analysis, ordering computed internally or provided by the user, Schur complement, assembled or elemental matrix), the symmetric permutation of the variables that MUMPS relies on is returned to the user in the mumps_par%SYM_PERM array.

mumps_par%**SYM_PERM** (integer pointer array, dimension `N`) is allocated internally and returned on the host processor on output to the analysis phase. It contains the permutation that was effectively computed during the analysis phase and that will serve as a basis for the numerical factorization. It is such that SYM_PERM(i) holds the position of variable i in the pivot order. For example, SYM_PERM(12)=2 means that variable 12 in the original matrix is the second variable to be eliminated in the pivot order. In case a Schur complement was requested (see `ICNTL(19)`), the returned permutation also includes the variables from the Schur complement, so that: `SYM_PERM`(`LISTVAR_SCHUR`(i))=N-`SIZE_SCHUR`+i, for $1 \leq i \leq$ `SIZE_SCHUR` (see also Subsection 5.14).

5.4.2 Given ordering (`ICNTL(7)`=1 and `ICNTL(28)`=1)

An ordering can optionally be provided by the user on input to the package. Even in this case, the analysis phase (`JOB`=1) must be performed before the numerical factorization. Note that this functionality is only available with the sequential analysis (`ICNTL(28)`=1).

mumps_par%**PERM_IN** (integer pointer array, dimension `N`) must be allocated and initialized by the user on the host before the sequential analysis phase (`JOB`=1, `ICNTL(28)`=1) when `ICNTL(7)`=1. Although the input matrix can be provided in assembled or elemental format (see `ICNTL(5)`), PERM_IN always defines the elimination order of the variables, not the elimination order of the elements. The user should define PERM_IN such that PERM_IN(i), i=1, ..., N holds the position of variable i in the pivot order. For example, PERM_IN(12)=2 indicates that variable 12 in the original matrix is the second variable to be eliminated in the pivot order. In case a Schur complement is requested (`ICNTL(19)`=1,2,3), the permutation should also include the variables from the Schur complement, so that: `PERM_IN`(`LISTVAR_SCHUR`(i))=N-`SIZE_SCHUR`+i, for $1 \leq i \leq$ `SIZE_SCHUR` (see Subsection 5.14). In case of parallel analysis (`ICNTL(28)`=2), `PERM_IN` is ignored. The input matrix can be centralized or distributed (see `ICNTL(18)`).

Remark that, in case of given ordering, although `PERM_IN` is used, `SYM_PERM` (see Subsection 5.4.1) will generally differ from `PERM_IN` because MUMPS takes some freedom to reorganize the order of the computations for locality and efficiency. However, `PERM_IN` and `SYM_PERM` are equivalent orderings in terms of estimated factor size and estimated number of operations for the factorization.

5.5 Post-processing: iterative refinement

It is possible to improve the accuracy of the solution using an iterative refinement procedure, thanks to the control parameter `ICNTL(10)`. The iterative refinement procedure can stop either when a stopping criterion is satisfied, or after a fixed number of steps. Algorithm 2 provides the iterative refinement procedure with a stopping criterion, as implemented in MUMPS. In that case, the stopping criterion is based on the backward errors ω_1 and ω_2, as defined in Section 3.3.2.

Algorithm 3 can also be used in order to perform a fixed number of steps of iterative refinement, without convergence test. In fact, it has been shown [21] that with only two to three steps of iterative refinement the solution can often be significantly improved.

Note that the iterative refinement method may diverge. In case of iterative refinement with a fixed number of steps, the final solution may then be worse than the initial solution. On the other hand, in case of divergence with Algorithm 2 at a given iteration, iterative refinement stops and the solution x is overwritten by the previous iterate.

ICNTL(10) applies iterative refinement to improve the computed solution.

Phase: accessed by the host during the solve phase.

Let x be the initial solution of $Ax = b$
Compute residual $r = b - Ax$
Compute the associated backward errors ω_1 and ω_2 (see Subsection 3.3.2)
$i = 0$
while $\omega_1 + \omega_2 \geq \alpha$ and convergence is not too slow and $i \leq$ IR_steps **do**
   Solve $A\Delta x = r$ using the computed factorization
   $x = x + \Delta x$
   $r = b - Ax$
   Compute backward errors ω_1 and ω_2
   $i = i + 1$
end while

Algorithm 2: Iterative refinement. At each step, backward errors are computed and compared to α, the *stopping criterion* (see CNTL(2)). The number of steps performed is limited to *IR_steps* (= ICNTL(10)).

Let x be the initial solution of $Ax = b$
Compute residual $r = b - Ax$
for $i = 1$ to IR_Steps **do**
   Solve $A\Delta x = r$ using the computed factorization
   $x = x + \Delta x$
   $r = b - Ax$
end for

Algorithm 3: Iterative refinement with a fixed number of steps IR_Steps (= |ICNTL(10)|).

Possible variables/arrays involved: NRHS

Possible values :

< 0 : Fixed number of steps of iterative refinement. No stopping criterion is used.

  0 : No iterative refinement.

> 0 : Maximum number of steps of iterative refinement. A stopping criterion is used, therefore a test for convergence is done at each step of the iterative refinement algorithm.

Default value: 0 (no iterative refinement)

Related parameters: CNTL(2)

Incompatibility: if ICNTL(21)=1 (solution kept distributed) or if ICNTL(32)=1 (forward elimination during factorization), or if NRHS>1 (multiple right hand sides), then ICNTL(10) is treated as 0.

Remarks: Note that if ICNTL(10) < 0, |ICNTL(10)| steps of iterative refinement are performed, without any test of convergence (see Algorithm 3). This means that the iterative refinement may diverge, that is the solution instead of being improved may be worse from an accuracy point of view. But it has been shown [21] that with only two to three steps of iterative refinement the solution can often be significantly improved. So if the convergence test should not be done we recommend to set ICNTL(10) to -2 or -3.

Note also that it is not necessary to activate the error analysis option (ICNTL(11) = 1,2) to be able to run the iterative refinement with stopping criterium (ICNTL(10) > 0). However, since the backward errors ω_1 and ω_2 have been computed, they are still returned in RINFOG(7) and RINFOG(8), respectively.

It must be noticed that iterative refinement with stopping criterium (ICNTL(10) > 0) will stop when

1. either the requested accuracy is reached ($\omega_1 + \omega_2 <$ `CNTL(2)`)

2. or when the convergence rate is too slow ($\omega_1 + \omega_2$ does not decrease by at least a factor of 5)

3. or when exactly ICNTL(10) steps have been performed.

In the first two cases the number of iterative refinement steps (`INFOG(15)`) may be lower than `ICNTL(10)`.

5.6 Post-processing: error analysis

`MUMPS` enables the user to perform classical error analysis based on the residuals. We calculate an estimate of the sparse backward error using the theory and metrics developed in [21] (see Subsection 3.3.2).

If `ICNTL(11)` = 2, main statistics are computed:

- $\omega_1 =$ RINFOG(7)

- $\omega_2 =$ RINFOG(8)

- the infinite norm of the input matrix: $\|A\|_\infty$ or $\|A^T\|_\infty =$ RINFOG(4)

- the infinite norm of the computed solution $\bar{x}$: $\|\bar{x}\|_\infty =$ RINFOG(5)

- the scaled residual: $\frac{\|A\bar{x}-b\|_\infty}{\|A\|_\infty \|\bar{x}\|_\infty} =$ RINFOG(6)

If `ICNTL(11)` = 1, in addition to the above statistics, the condition numbers for the linear system (not just the matrix) and an upper bound of the forward error of the computed solution are also returned (see Subsection 3.3.2):

- $cond_1 =$ RINFOG(10)

- $cond_2 =$ RINFOG(11)

- $\frac{\|\delta\bar{x}\|_\infty}{\|\bar{x}\|_\infty} \leq \omega_1\, cond_1 + \omega_2\, cond_2 =$ RINFOG(9)

Note that the error analysis in the case of `ICNTL(11)` = 1 is significantly more costly than the solve phase itself.

ICNTL(11) computes statistics related to an error analysis of the linear system solved ($\mathbf{Ax} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ (see `ICNTL(9)`)).

Phase: accessed by the host and only during the solve phase.

Possible variables/arrays involved: NRHS

Possible values :

 0 : no error analysis is performed (no statistics).

 1 : compute all the statistics (very expensive).

 2 : compute main statistics (norms, residuals, componentwise backward errors), but not the most expensive ones like (condition number and forward error estimates).

Values different from 0, 1, and 2 are treated as 0.

Default value: 0 (no statistics).

Incompatibility: if NRHS > 1, if `ICNTL(32)`=1 (forward elimination during factorization), or if `ICNTL(21)`=1 (solution kept distributed) then error analysis is not performed and `ICNTL(11)` is treated as 0

Related parameters: `ICNTL(9)`

Remarks: The computed statistics are returned in various informational parameters, see also Subsection 3.3:

 – If ICNTL(11)= 2, then the infinite norm of the input matrix ($\|A\|_\infty$ or $\|A^T\|_\infty$ in RINFOG(4)), the infinite norm of the computed solution ($\|\bar{x}\|_\infty$ in RINFOG(5)), and the scaled residual $\frac{\|A\bar{x}-b\|_\infty}{\|A\|_\infty \|\bar{x}\|_\infty}$ in RINFOG(6), a componentwise backward error estimate in RINFOG(7) and RINFOG(8) are computed.

– If ICNTL(11)= 1, then in addition to the above statistics also an estimate for the error in the solution in RINFOG(9), and condition numbers for the linear system in RINFOG(10) and RINFOG(11) are also returned.

If performance is critical, ICNTL(11) should be set to 0. If both performance is critical and statistics are requested, then ICNTL(11) should be set to 2. If ICNTL(11)=1, the error analysis is very costly (typically significantly more costly than the solve phase itself).

5.7   Out-of-core (`ICNTL(22)`)

The decision to use the disk to store the matrix of factors is controlled by `ICNTL(22)`. Only the value on the host node is significant.

ICNTL(22) controls the in-core/out-of-core (OOC) factorization and solve.

Phase: accessed by the host during the factorization phase.

Possible variables/arrays involved: `OOC_TMPDIR` and `OOC_PREFIX`

Possible values :

   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 Subsection 3.13).

Default value: 0 (in-core factorization)

Remarks: The variables `OOC_TMPDIR` and `OOC_PREFIX` are used to indicate the directory and the prefix, respectively, where to store the factors. They must be set after the initialization phase (`JOB` = -1) and before the factorization phase (`JOB`=2,4,5 or 6). Otherwise, `MUMPS` will use the `/tmp` directory and arbitrary file names. Note `MUMPS` accesses to the variables `OOC_TMPDIR` and `OOC_PREFIX` only during the factorization phase. Several files under the same directory and with the same prefix are created to store the factors. Their names contain a unique hash and `MUMPS` is in charge of keeping trace of them.

The files containing the factors will be deleted if a new factorization starts or when a termination phase (`JOB`=-2) is called.

Note that, in case of abnormal termination of an application calling `MUMPS` (for example, a termination of the calling process with a segmentation fault, or, more generally, a termination of the calling process without a call to `MUMPS` with `JOB`=-2), the files containing the factors are not deleted. It is then the user's responsibility to delete them, as shown in bold in the example below, where the application calling `MUMPS` is launched from a bash script and environment variables are used to define the OOC environment:

```
#!/bin/bash
export MUMPS_OOC_TMPDIR="/local/mumps_data/"
export MUMPS_OOC_PREFIX="job_myapp_"
mpirun -np 128 ./myapplication
# Suppress MUMPS OOC files in case of bad application termination
rm -f ${MUMPS_OOC_TMPDIR}/${MUMPS_OOC_PREFIX}*
```

mumps_par%**OOC_TMPDIR** (string) can be provided by the user (on each processor) to control the directory where the out-of-core files will be stored.

Note that it is also possible to provide the directory through environment variables. If `OOC_TMPDIR` is not defined, then `MUMPS` checks for the environment variable *MUMPS_OOC_TMPDIR*. If neither `OOC_TMPDIR` nor *MUMPS_OOC_TMPDIR* are defined, then the directory `/tmp` is attempted.

mumps_par%**OOC_PREFIX** (string) can be provided by the user (on each processor) to prefix the out-of-core files.

Note that it is also possible to provide the files prefix through environment variables. If `OOC_PREFIX` is not defined, then `MUMPS` checks for the environment variable *MUMPS_OOC_PREFIX*. If neither `OOC_PREFIX` nor *MUMPS_OOC_PREFIX* are defined, then `MUMPS` chooses the file names automatically.

5.8 Workspace parameters (`ICNTL(14)` and `ICNTL(23)`) and user workspace

The memory required to run the numerical phases (factorization and solve) is estimated during the analysis. The size of the workspace actually required during numerical factorization depends on the numerical characteristics of the matrix, and therefore on the numerical pivoting that may lead to extra storage, but also on algorithmic parameters such as the in-core/out-of-core strategies (`ICNTL(22)`), the memory relaxation parameter (`ICNTL(14)`) and the fact to discard the factor matrices during the factorization (`ICNTL(31)`).

Two main workarrays are allocated internally: IS and S (integer and real/complex workarray, respectively), that hold factors, active frontal matrices, and contribution blocks. In addition to these two large workarrays, other internal workarrays are used: 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 (note that these estimations take into account the value of the memory relaxation parameter `ICNTL(14)`):

Estimated memory for the **in-core strategy**:

- Estimated size in MegaBytes (millions of bytes) of the total working space locally requested by each processor `INFO(15)`

- the maximum and sum over all processors: `INFOG(16)` and `INFOG(17)`, respectively

- the size of the main real/complex workarray S: `INFO(8)`. If `INFO(8)` is negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray.

Estimated memory for the **out-of-core strategy**:

- Estimated size in MegaBytes (millions of bytes) of the total working space locally requested by each processor : `INFO(17)`

- the maximum and sum over all processors: `INFOG(26)` and `INFOG(27)` respectively

- the size of the main real/complex workarray S: `INFO(20)`. If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray.

As a first general approach, we advise the user to rely on the estimations provided during the analysis phase. If the user wants to/must increase the allocated workspace (typically, because of 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 user can modify the value of the memory relaxation parameter, `ICNTL(14)`, that is designed to control the increase with respect to the estimations performed during analysis, in the size of all (integer and real/complex) workspace allocated during the numerical phase.

- The user can explicitly control the memory used by the package providing in `ICNTL(23)` the size of the total memory that is allowed to be used internally.

We provide the definitions of `ICNTL(14)` and `ICNTL(23)` below:

ICNTL(14) corresponds to the percentage increase in the estimated working space.

Phase: accessed by the host both during the analysis and the factorization phases.

Default value: 20 (which corresponds to a 20 % increase).

Related parameters: `ICNTL(23)`

Remarks: When significant extra fill-in is caused by numerical pivoting, increasing ICNTL(14) may help.

ICNTL(23) 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.

Phase: accessed by the host at the beginning of the factorization phase and is only significant on the host.

Possible values :

   0 : each processor will allocate workspace based on the estimates computed during the analysis

 >0 : maximum size of the working memory in MegaBytes per working processor to be allocated

Default value: 0

Related parameters: `ICNTL(14)`

Remarks: 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 for the factorization phase a workspace based on the estimates computed during the analysis (`INFO(17)` that is the estimate of the sum over all processors) 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 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 the maximum over all processors: `INFOG(16)` in the in-core case, or `INFOG(26)` in the out-of-core case.

Another possibility is that the user provides the real/complex workarray instead of using the internal main real/complex workarray S. The user workarray must be larger than `INFO(8)` (in-core factorization) or `INFO(20)` (out-of-core factorization), and a pointer array that points to that workspace must be provided. In this case, the value of `ICNTL(23)` is ignored.

We describe below the two parameters associated to this functionality:

mumps_par%**LWK_USER** (integer) is local to each processor if `PAR`=1, and on all processors except the host if `PAR`=0. It is accessed at the beginning of the numerical phases of `MUMPS`. Its default value is 0. At the beginning of the numerical phases, if the user sets LWK_USER to a nonzero value then LWK_USER will be define the size of the pointer array `WK_USER`. If negative, -LWK_USER is a lower bound of the number of entries in millions of the pointer array `WK_USER` so that $abs(\text{LWK\_USER}) \times 10^6 \leq size(\text{WK\_USER})$ must hold.

If the numerical phases are in-core (`ICNTL(22)`=0), we recommend the user to set LWK_USER larger than `INFO(8)`, otherwise an error with code `--9` may occur. Moreover, the value of LWK_USER must not be modified between factorization and subsequent solution phases.

If the numerical phases are out-of-core (`ICNTL(22)`=1), we recommend LWK_USER to be larger than `INFO(20)`. In this case, the user can reduce the value for LWK_USER between the factorization phase and the solve phase.

mumps_par%**WK_USER** is a real/complex pointer array that can point to the workspace provided by the user. It is only accessed by `MUMPS` when `LWK_USER` has been set by the user to a non-zero value. In that case, `MUMPS` will avoid the internal allocation of the main real/complex workarray S and use WK_USER instead.

Note that the type of WK_USER should follow the arithmetic: single precision for `SMUMPS`, double precision for `DMUMPS`, single complex for `CMUMPS`, and double complex for `ZMUMPS`.

If the factorization is in-core (`ICNTL(22)`=0), then WK_USER should not be modified between factorization and solution phases of `MUMPS`.

5.9   Null pivot row detection

It is possible to detect null pivots during the factorization by setting `ICNTL(24)` = 1. A pivot is considered as null if it is smaller than a threshold defined by `CNTL(3)` .

At the end of the factorization mumps_par%`INFOG(28)` will contain the deficiency of the initial matrix and the array `PIVNUL_LIST`(1:`INFOG(28)`) will contain, on the host, the row indices corresponding to the null pivots, if `INFOG(28)` $\neq 0$.

In order to be able to perform a solve step, the null pivot may be replaced by a huge fixation or by 1 (in this latter case the elements of the columns under the pivot are set to 0). The choice of the fixation value can be done with `CNTL(5)`. In this way we will be able to compute one solution among the possible solutions of the deficient system $AX = B$ (`ICNTL(25)` $= 0$) or a part of the null vectors (that is the vectors solving $AX = 0$, `ICNTL(25)` $\neq 0$) associated to these null pivots.

ICNTL(24) controls the detection of "null pivot rows".

> *Phase*: accessed by the host during the factorization phase
>
> *Possible variables/arrays involved*: `PIVNUL_LIST`
>
> *Possible values* :
>
> > 0: Nothing done. A null pivot will result in error `INFO(1)`=−−10.
> > 1: Null pivot row/column detection.
>
> Other values are treated as 0.
>
> *Default value*: 0 (no null pivot detection)
>
> *Related parameters*: `CNTL(3)`, `CNTL(5)`, `ICNTL(13)`, `ICNTL(25)`
>
> *Remarks*:
>
> `CNTL(3)` is used to compute the threshold to decide if a pivot row is "null".
>
> Null pivot rows are modified to enable the solution phase to provide one solution among the possible solutions of the numerically deficient matrix. The parameter `CNTL(5)` defines the fixation of null pivots.
>
> 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, 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)`).
>
> Note that when ScaLAPACK is applied on the root node (see `ICNTL(13)` $= 0$), then exact null pivots on the root will stop the factorization (`INFO(1)`=−−10) while if tiny pivots are present on the root node the ScaLAPACK routine will factorize the root matrix. Computing the root node factorization sequentially (this can be forced by setting `ICNTL(13)` to 1) will help with the correct detection of null pivots but may degrade performance.

> Related data:

mumps_par%`INFOG(28)` (integer) is set to the number of null pivots detected during the factorization step. .

mumps_par%**PIVNUL_LIST** (integer array, dimension `N`) is set during the factorization phase to the row indices corresponding to the null pivots. It is also accessed during the solve phase.

5.10   Discard matrix factors (`ICNTL(31)`)

In some cases the user may want to discard one or both factors during factorization. This can be done using the `ICNTL(31)` control parameter.

ICNTL(31) indicates which factors may be discarded during the factorization.

> *Phase*: accessed by the host during the analysis phase.
>
> *Possible values* :

0 : the factors are kept during the factorization phase except in the case of the out-of-core factorization of unsymmetric matrices when the forward elimination is performed during factorization (`ICNTL(32)` = 1). In this case, since it will not be used during the solve phase, the **L** factor is discarded: it is not written to disk.

1 : all factors are discarded during the factorization phase. 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 this case, the memory allocated for the factorization will rely on the out-of-core estimates (and factors will not be written to disk).

2 : this setting is meaningful only for unsymmetric matrices and has no impact on symmetric matrices: only the **U** factor is kept after factorization so that exclusively a backward substitution is possible during the solve phase (`JOB`=3). This can be useful when:
−the user is only interested in the computation of a null space basis (see `ICNTL(25)`) during the solve phase, or
−the forward elimination is performed during the factorization (`ICNTL(32)`=1). Note that for unsymmetric matrices in out-of-core environments, if the forward elimination is performed during the factorization (`ICNTL(32)` = 1) then the **L** factor is always discarded during factorization. In this case (`ICNTL(32)` = 1), both `ICNTL(31)` = 0 and `ICNTL(31)` = 2 have the same behaviour.

Other values are treated as 0.

Default value: 0 (the factors are kept during the factorization phase in order to be able to solve the system).

Incompatibility: `ICNTL(31)` = 2 is not meaningful for symmetric matrices.

Related parameters: `ICNTL(32)`, forward elimination during factorization, `ICNTL(33)`, computation of the determinant, `ICNTL(25)` computation of a null space basis, `ICNTL(22)` out-of-core factors.

Remarks: For unsymmetric matrices, MUMPS currently discards **L** factors only in the out-of-core case (even when `ICNTL(32)` = 2). In a future version, discarding the **L** factor in the in-core case as well when `ICNTL(32)`=2 (or when `ICNTL(31)`=0 and `ICNTL(32)`=1) may lead to a memory reduction during the factorization.

5.11 Computation of the determinant (`ICNTL(33)`)

The user interested in computing the determinant of the matrix **A** can use the control parameter `ICNTL(33)`. See Subsection 3.14 for details on how it will be computed.

ICNTL(33) computes the determinant of the input matrix.

Phase: accessed by the host during the factorization phase.

Possible values :

0 : the determinant of the input matrix is not computed.

$\neq 0$: computes the determinant of the input matrix. The determinant is obtained by computing $(a + ib) \times 2^c$ where $a =$`RINFOG(12)`, $b =$`RINFOG(13)` and $c =$ `INFOG(34)`. In real arithmetic b=`RINFOG(13)` is equal to 0.

Default value: 0 (determinant is not computed)

Related parameters: `ICNTL(31)`

Remarks: In case a Schur complement was requested (see `ICNTL(19)`), elements of the Schur complement are excluded from the computation of the determinant so that the determinant is that one of matrix $A_{1,1}$ (using notations of Subsection 3.16).

Although we recommend to compute the determinant on non-singular matrices, null pivots (`ICNTL(24)`) and static pivots (`CNTL(4)`) are excluded from the determinant so that a non-zero determinant is still returned on singular or near-singular matrices. This determinant is then not unique and will depend on which equations were excluded.

Furthermore, we recommend to switch off scaling (`ICNTL(8)`) in such cases. If not (`ICNTL(8)` $\neq 0$), we describe in the following the current behaviour of the package:

– if static pivoting (`CNTL(4)`) is activated: all entries of the scaling arrays `ROWSCA` and `COLSCA` are currently taken into account in the computation of the determinant.

– if the null pivot detection (`ICNTL(24)`) is activated, then entries of `ROWSCA` and `COLSCA` corresponding to pivots in `PIVNUL_LIST` are excluded from the determinant so that

  ∗ for symmetric matrices (`SYM`=1 or 2), the returned determinant correctly corresponds to the matrix excluding rows and columns of `PIVNUL_LIST`.

  ∗ for unsymmetric matrices (`SYM`=0), scaling may perturb the value of the determinant in case off-diagonal pivoting has occurred (`INFOG(12)` $\neq 0$).

Note that if the user is interested in computing only the determinant, we recommend to discard the factors during factorization `ICNTL(31)`.

5.12   Forward elimination during factorization (`ICNTL(32)`)

This option makes much sense when the factors have to be used only once or in an out-of-core context (`ICNTL(22)`=1), where loading the factors from disk during the solution phase (`JOB`=3), both during the forward (Equation (3)) and backward (Equation (4)) substitutions, can be particularly costly.

Factorization type	Phase	Without Forward during factorization ICNTL(32) = 0	With forward during factorization ICNTL(32) = 1
LU	Factorization phase	$A = LU$	$A = LU$ **Solve** $Ly = b$
	Solve phase	**Solve** $Ly = b$ Solve $Ux = y$	Solve $Ux = y$
LDL^T	Factorization phase	$A = LDL^T$	$A = LDL^T$ **Solve** $LDy = b$
	Solve phase	**Solve** $LDy = b$ Solve $L^T x = y$	Solve $L^T x = y$

Note that for unsymmetric matrices, if the forward elimination is performed during factorization, the **U** factor may be discarded (see `ICNTL(31)`). In the symmetric LDL^T case, the **L** factor must always be kept in order to be able to solve $L^T x = y$.

ICNTL(32)   performs the forward elimination of the right-hand sides (Equation (3)) during the factorization (`JOB`=2).

Phase: accessed by the host during the analysis phase.

Possible variables/arrays involved: RHS, `NRHS`, `LRHS`, and possibly `REDRHS`, `LREDRHS` when `ICNTL(26)`=1

Possible values :

  0: standard factorization not involving right-hand sides.
  1: forward elimination (Equation (3)) of the right-hand side vectors is performed during factorization (`JOB`=2). The solve phase (`JOB`=3) will then only involve backward substitution (Equation (4)).

Other values are treated as 0.

Default value: 0 (standard factorization)

Related parameters: `ICNTL(31)`,`ICNTL(26)`

Incompatibility: This option is incompatible with sparse right-hand sides (`ICNTL(20)`=1,2,3), with the solution of the transposed system (`ICNTL(9)` $\neq$ 1), and with the computation of entries of the inverse (`ICNTL(30)`).

Furthermore, iterative refinement (`ICNTL(10)`) and error analysis (`ICNTL(11)`) are disabled. Finally, the current implementation imposes that all right-hand sides are processed in one pass during the backward step. Therefore, the blocking size (`ICNTL(27)`) is ignored.

Remarks: The right-hand sides must be dense to use this functionality: RHS, `NRHS`, and `LRHS` should be provided as described in Subsection 5.13.1. They should be provided at the beginning of the factorization phase (`JOB`=2) rather than at the beginning of the solve phase (`JOB`=3).

For unsymmetric matrices if the forward elimination is performed during factorization (`ICNTL(32)` = 1), the L factor (see `ICNTL(31)`) may be discarded to save space. In fact for unsymmetric matrices in out-of-core environments, if the forward elimination is performed during the factorization (`ICNTL(32)` = 1) then the L factors are always discarded during factorization even when `ICNTL(31)` = 0.

We advise to use this option only for a reasonable number of dense right-hand side vectors because of the additional associated storage required when this option is activated and the number of right-hand sides is large compared to `ICNTL(27)`.

5.13   Right-hand side and solution vectors/matrices

`MUMPS` can solve the systems $\mathbf{AX} = \mathbf{B}$ or $\mathbf{A}^T\mathbf{X} = \mathbf{B}$ where $\mathbf{X}, \mathbf{B} \in \mathbb{R}^{n \times nrhs}$. The $\mathbf{B}$ matrix is referred to as the right-hand side and the $\mathbf{X}$ matrix to as the solution.

   `MUMPS` gives the option to input the right-hand side matrix $\mathbf{B}$ in dense or in sparse format but always centralized on the host processor, and to output the solution matrix $\mathbf{X}$ centralized or distributed, but always in dense format. Sparsity of the right-hand side can be exploited to accelerate the solution phase [15, 16, 41, 35]. Note that the first step of the solution phase involves the distribution (scatter step) of the right-hand side onto the processors. The cost of scatter can be highly reduce in when right-hand sides are sparse and provided in a sparse format.

   Moreover, `MUMPS` can optionally compute some entries of the inverse $\mathbf{A}^{-1}$ solving the system with a particular sparse right-hand side $\mathbf{B}$ (see Subsection 5.13.3).

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

ICNTL(20) determines the format (dense or sparse) of the right-hand side.

   Phase: accessed by the host during the solve phase.

   Possible variables/arrays involved: RHS, `NRHS`, `LRHS`, `IRHS_SPARSE`, `RHS_SPARSE`, `IRHS_PTR` and `NZ_RHS`.

   Possible values :

   0 : the right-hand side is in dense format in the structure component RHS, `NRHS`, `LRHS` (see Subsection 5.13.1)

   1,2,3 : the right-hand side is in sparse format in the structure components `IRHS_SPARSE`, `RHS_SPARSE`, `IRHS_PTR` and `NZ_RHS`.

      1 : The decision of exploiting sparsity of the right-hand side to accelerate the solution phase is done automatically.

      2 : Sparsity of the right-hand side is NOT exploited to improve solution phase.

      3 : Sparsity of the right-hand side is exploited during solution phase.

   Values different from 0, 1, 2 or 3 are treated as 0. For a sparse right-hand side, the recommended value is 1.

   Default value: 0 (dense right-hand sides)

   Incompatibility: When ICNTL(20)=0 (dense right-hand side) and `NRHS` > 1 (multiple right-hand side) the functionalities related to iterative refinement (`ICNTL(10)`) and error analysis (`ICNTL(11)`) are currently disabled.

With sparse right-hand sides (ICNTL(20)=1,2,3), the forward elimination during the factorization (`ICNTL(32)`=1) is not currently available.

Remarks: For details on how to set the input parameters see Subsection 5.13.1 and Subsection 5.13.2. Please note that duplicate entries in the sparse right-hand sides are summed.

ICNTL(21) determines the distribution (centralized or distributed) of the solution vectors.

Phase: accessed by the host during the solve phase.

Possible variables/arrays involved: RHS, `ISOL_loc` and `SOL_loc`, `LSOL_loc`

Possible values :

- 0 : the solution vector is assembled and stored in the structure component RHS (gather phase), that must have been allocated earlier by the user (see Subsection 5.13.4).
- 1 : the solution vector is kept distributed 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 (see Subsection 5.13.5).

Values different from 0 and 1 are currently treated as 0.

Default value: 0 (assembled centralized format)

Incompatibility: If the solution is kept distributed, error analysis and iterative refinement (controlled by `ICNTL(10)` and `ICNTL(11)`) are not applied.

5.13.1 Dense right-hand side (`ICNTL(20)`=0)

In this case, the matrix **B** of size n x $nrhs$ is input in a one dimensional array of size $lrhs$ x $nrhs$ where the leading dimension $lrhs$ must be $\geq n$, the dimension of the matrix **A**.

The following components of the MUMPS structure should be allocated by the user on the host before a call to MUMPS with JOB= 3, 5, or 6 (call including the solve) if forward and backward elimination are both computed during the solve (`ICNTL(32)`=0), or before a call to MUMPS with JOB= 2, 4 (call including the factorization) if the forward elimination is computed during the factorization (`ICNTL(32)`=1).

mumps_par%**RHS** (**real/complex** pointer array, dimension LRHS×NRHS) is a **real** (**complex** in the complex version) array.

On entry RHS(i+(k-1)× LRHS) must hold the i-th component of the kth column of the right-hand side matrix ($1 \leq k \leq$ NRHS) of the equations being solved.

On exit, if the solution matrix has to be centralized (`ICNTL(21)`=0), then RHS(i+(k-1)×LRHS) will hold the i-th component of the kth column of the solution matrix, $1 \leq k \leq$ NRHS.

Otherwise, if the solution matrix has to be distributed (`ICNTL(21)`=1), on exit to the package, RHS will not contain any significant data for the user, even if it may have been modified.

mumps_par%**NRHS** (integer) is an optional parameter that should be set by the user to the number of right-hand side vectors. Otherwise, the value 1 is assumed.

mumps_par%**LRHS** (integer) is an optional parameter that should be set by the user, in the case where NRHS is set by the user. In this case, it must hold the leading dimension of the array RHS and should be greater than or equal to N (the matrix dimension). Otherwise, a single-column right-hand side is assumed and LRHS is not accessed.

5.13.2 Sparse right-hand side (`ICNTL(20)`=1,2,3)

If the user wants to compute the solution with sparse right-hand sides, the right-hand side matrix should be input as a sparse matrix in column format. Sparsity of the right-hand side can be exploited to accelerate the solution phase (see [41, 35]). Since exploiting sparsity requires some extra preprocessing it can be switched of by the user setting `ICNTL(20)` to 2.

In the following, an example of a 4x2 matrix **B** with 5 nonzeros is provided.

$$\mathbf{B} = \begin{pmatrix} a_{11} & \\ & a_{22} \\ a_{31} & a_{32} \\ a_{41} & \end{pmatrix}$$

total nonzeros in **B**	NZ_RHS	= 5					
number of columns **B** (n. of rhs vectors)	NRHS	= 2					
pointers to the columns	IRHS_PTR	$[1 : NRHS + 1]$	=	1	4	6	
array of row indices	IRHS_SPARSE	$[1 : NZ\_RHS]$	=	1	3	4	2	3
array of values	RHS_SPARSE	$[1 : NZ\_RHS]$	=	a_{11}	a_{31}	a_{41}	a_{22}	a_{32}

The following input parameters should be defined on the host only before a call to MUMPS including the solve phase (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** pointer array, dimension NZ_RHS) should hold the numerical values of the non-zero entries of each right-hand side vector. This means that the B matrix should be input by columns.

mumps_par%**IRHS_SPARSE** (integer pointer array, 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** (integer pointer array, dimension NRHS+1) 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.

mumps_par%**RHS** (**real/complex** pointer array, dimension LRHS×NRHS) must be allocated by the user on the host if the output solution should be centralized (ICNTL(21)=0). On exit from a call to MUMPS it will hold the centralized solution (ICNTL(21)=0).

5.13.3 A particular case of sparse right-hand side: computing entries of $\mathbf{A}^{-1}$ (ICNTL(30)=1)

It is possible to compute some selected entries of the inverse matrix $\mathbf{A}^{-1}$ (see Subsection 3.15) using the control parameter ICNTL(30).

Let us consider the example below, in which

$$\mathbf{A}^{-1} = \begin{pmatrix} a_{11}^{-1} & a_{12}^{-1} & a_{13}^{-1} & a_{14}^{-1} \\ a_{21}^{-1} & a_{22}^{-1} & a_{23}^{-1} & a_{24}^{-1} \\ a_{31}^{-1} & a_{32}^{-1} & a_{33}^{-1} & a_{34}^{-1} \\ a_{41}^{-1} & a_{42}^{-1} & a_{43}^{-1} & a_{44}^{-1} \end{pmatrix}$$

denotes the inverse of the matrix $\mathbf{A}$.

We would like to compute the boldface elements:

$$\mathbf{A}^{-1} = \begin{pmatrix} \mathbf{a_{11}^{-1}} & a_{12}^{-1} & a_{13}^{-1} & a_{14}^{-1} \\ a_{21}^{-1} & a_{22}^{-1} & a_{23}^{-1} & a_{24}^{-1} \\ \mathbf{a_{31}^{-1}} & \mathbf{a_{32}^{-1}} & a_{33}^{-1} & a_{34}^{-1} \\ a_{41}^{-1} & a_{42}^{-1} & a_{43}^{-1} & \mathbf{a_{44}^{-1}} \end{pmatrix}$$

On input, the following parameters should be allocated and initialized:

total entries $\mathbf{A}^{-1}$ to be computed	NZ_RHS	= 4						
number of columns $\mathbf{A}^{-1}$	NRHS	= N	= 4					
pointers to the columns	IRHS_PTR	$[1 : NRHS + 1]$	=	1	3	4	4	5
array of row indices	IRHS_SPARSE	$[1 : NZ\_RHS]$	=	1	3	3	4	

Note that column 3 will be considered as empty, because no elements have to be computed. The following parameter should be allocated, but not initialized:

array of values	RHS_SPARSE	$[1 : NZ\_RHS]$

On output, the following parameters will hold the requested entries:

total entries $\mathbf{A}^{-1}$ to be computed	NZ_RHS	= 4						
number of columns $\mathbf{A}^{-1}$	NRHS	= N	= 4					
pointers to the columns	IRHS_PTR	$[1 : NRHS + 1]$	=	1	3	4	4	5
array of row indices	IRHS_SPARSE	$[1 : NZ\_RHS]$	=	1	3	3	4	
array of values	RHS_SPARSE	$[1 : NZ\_RHS]$	=	a_{11}^{-1}	a_{31}^{-1}	a_{32}^{-1}	a_{44}^{-1}	

ICNTL(30) computes a user-specified set of entries in the inverse $\mathbf{A}^{-1}$ of the original matrix.

Phase: accessed during the solution phase.

Possible variables/arrays involved: NZ_RHS, NRHS, RHS_SPARSE, IRHS_SPARSE, IRHS_PTR

Possible values :

   0: no entries in $\mathbf{A}^{-1}$ are computed.

   1: computes entries in $\mathbf{A}^{-1}$.

Other values are treated as 0.

Default value: 0 (no entries in $\mathbf{A}^{-1}$ are computed)

Incompatibility: Error analysis and iterative refinement will not be performed, even if the corresponding options are set (ICNTL(10) and ICNTL(11)). 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)). Option to compute solution using $\mathbf{A}$ or $\mathbf{A}^T$ (ICNTL(9)) is meaningless and thus ignored.

Related parameters: ICNTL(27)

Remarks: 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). Larger ICNTL(27) values will most likely decrease the amount of factor accesses, enable more parallelism and thus reduce the solution time [45, 41, 15].

The user must specify on input to a call of the solve phase in the arrays IRHS_PTR and IRHS_SPARSE the target entries. The array RHS_SPARSE should be allocated but not initialized. Note that since selected entries of the inverse of the matrix are requested, NRHS must be set to N. On output the arrays IRHS_PTR, IRHS_SPARSE and RHS_SPARSE will hold the requested entries. If duplicate target entries are provided then duplicate solutions will be returned.

When entries of $\mathbf{A}^{-1}$ are requested (ICNTL(30) = 1), mumps_par%**RHS** needs not be allocated.

5.13.4   Centralized solution (ICNTL(21)=0)

The solution vector $\mathbf{X}$ can be returned centralized on the host in both cases: dense or sparse right-hand side vectors. The matrix $\mathbf{X}$ will be returned as a dense vector in the array mumps_par%RHS. For this reason the mumps_par%RHS should be allocated even when the right-hand side matrix $\mathbf{B}$ is input in sparse format.

5.13.5 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 (`ICNTL(21)=1`). The solution should then be exploited in its distributed form by the user application.

Note that this option can be used only with `JOB`=3 and should not be used with `JOB`= 5 or 6, because some parameters needed for this option must be set using information output by the factorization.

The following input parameters 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).

mumps_par%**SOL_loc** (**real/complex** pointer array, dimension `LSOL_loc`× `NRHS` where `NRHS` is either equal to 1 or corresponds to the value provided by the user in `NRHS` on the host) must be allocated by the user between the factorization and solve steps . Its leading dimension `LSOL_loc` should be larger than or equal to `INFO(23)`, that is returned by 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 kth 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)`, that is returned by the factorization phase.

mumps_par%**ISOL_loc** (integer pointer array, of dimension at least `INFO(23)`, that is returned by the factorization phase) must be allocated by the user between the factorization and solve steps.

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.

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 (`ICNTL(10)` and `ICNTL(11)`) are currently not available.

5.14 Schur complement with reduced or condensed right-hand side (`ICNTL(19)` **and** `ICNTL(26)`)

`MUMPS` gives the possibility to perform the partial factorization of the complete matrix and to return the Schur matrix, that is the part of the matrix still to be factorized. The Schur matrix will be returned as a full matrix, distributed in different ways (see Subsection 5.14.1, Subsection 5.14.3 and Subsection 5.14.2).

ICNTL(19) computes the Schur complement matrix.

Phase: accessed by the host during the analysis phase.

Possible variables/arrays involved: `SIZE_SCHUR`, `LISTVAR_SCHUR`, `NPROW`, `NPCOL`, `MBLOCK`, `NBLOCK`, `SCHUR`, `SCHUR_MLOC`, `SCHUR_NLOC`, and `SCHUR_LLD`

Possible values :

0 : complete factorization. No Schur complement is returned.

1 : the Schur complement matrix will be returned centralized by rows on the host after the factorization phase. On the host before the analysis phase, the user must set the integer variable `SIZE_SCHUR` to the size of the Schur matrix, the integer pointer array `LISTVAR_SCHUR` to the list of indices of the Schur matrix.

2 or 3 : the Schur complement matrix will be returned distributed by columns: the Schur will be returned on the slave processors in the form of a 2D block cyclic distributed matrix (ScaLAPACK style) after factorization. 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 Subsection 5.14). On the host before the analysis phase, the user must set the integer variable `SIZE_SCHUR` to the size of the Schur matrix, the integer pointer array `LISTVAR_SCHUR` to the list of indices of the Schur matrix.

The integer variables `NPROW`, `NPCOL`, `MBLOCK`, `NBLOCK` may also be defined (default values will otherwise be provided).

Values not equal to 1, 2 or 3 are treated as 0.

Default value: 0 (complete factorization)

Incompatibility: 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.

Related parameters: `ICNTL(7)`

Remarks: If the ordering is given (`ICNTL(7)`=1) then the following property should hold: `PERM_IN(LISTVAR_SCHUR(i)) = N-SIZE_SCHUR+i`, for i=1,`SIZE_SCHUR`.

Note that, in order to have a centralized Schur complement matrix by columns (see Subsection 5.14.3), it is possible (and recommended) to use a particular case of the distributed Schur complement (`ICNTL(19)`=2 or 3), where the Schur complement is assigned to only one processor (`NPCOL` × `NPROW` = 1).

If `ICNTL(19)` = 1,2,3 the user should give on input on the host before the analysis phase the following parameters:

mumps_par%**SIZE_SCHUR** (integer) must be initialized to the number of variables defining the Schur complement. 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`.

mumps_par%**LISTVAR_SCHUR** (integer pointer array, dimension `SIZE_SCHUR`) must be allocated and initialized by the user so that LISTVAR_SCHUR(i), i=1, ..., SIZE_SCHUR must hold the ith variable of the Schur complement matrix. It is accessed during analysis (`JOB`=1) and it is not altered by `MUMPS`.

If a given ordering (Subsection 5.4.2, `ICNTL(7)` =1) is set by the user, the permutation should also include the variables of the Schur complement, so that: `PERM_IN(LISTVAR_SCHUR(i))`=N-`SIZE_SCHUR`+i, for $1 \leq i \leq$`SIZE_SCHUR`.

5.14.1 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 `SCHUR` argument. If a centralized Schur complement is required, we refer the user to the Subsection 5.14.3 "Centralized Schur complement stored by columns" 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 `PAR`) before the factorization phase. 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.

5.14.2 Distributed Schur complement (`ICNTL(19)`=2 or 3)

`MUMPS` gives the possibility to output the Schur complement matrix distributed onto the processors (`ICNTL(19)` = 2,3) using a 2D block cyclic distribution (please refer to [22] (for example) for the notion of grid of processors and 2D block cyclic distributions) stored by columns.

For symmetric matrices, if `ICNTL(19)` =2, only the lower part of the Schur matrix is generated, otherwise, if `ICNTL(19)` =3, the complete Schur matrix is generated.

For unsymmetric matrices `MUMPS` always provides the complete Schur matrix, so that `ICNTL(19)`=2 and `ICNTL(19)`=3 have the same effect.

On entry to the analysis phase (`JOB` = 1), the following parameters should be defined on the host:

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. 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 `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. We briefly describe here the meaning of the four above parameters in a 2D block cyclic distribution:

- 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 MAX(1,NUMROC(`SIZE_SCHUR`, `MBLOCK`, *myrow*, 0, `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 *myid* / `NPCOL`,
    - if `PAR` = 0 *myrow* is equal to $(myid - 1)$ / `NPCOL`.

Note that an upperbound of the minimum value of leading dimension (`SCHUR_LLD` defined below) is equal to ((`SIZE_SCHUR`+`MBLOCK`-1)/`MBLOCK`+`NPROW`-1)/`NPROW`*`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 NUMROC(`SIZE_SCHUR`, `NBLOCK`, *mycol*, 0, `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 MOD(*myid*, `NPCOL`),
    - if `PAR` = 0 *mycol* is equal to MOD($myid - 1$, `NPCOL`).

On entry to the factorization phase (`JOB` = 2), the user should give on input the following components of the structure:

mumps_par%**SCHUR_LLD** (integer) should be set to 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** (**real/complex** one-dimensional pointer array) 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). Its size should be at least equal to `SCHUR_LLD` × (`SCHUR_NLOC` - 1) + `SCHUR_MLOC`, where `SCHUR_MLOC`, `SCHUR_NLOC`, and `SCHUR_LLD` have been defined above.

On exit from 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`.

Note that if `ICNTL(19)`=3 and the Schur is symmetric (`SYM`=1 or 2), then the constraint mumps_par%`MBLOCK` = mumps_par%`NBLOCK` should hold.

Note that setting `NPCOL` × `NPROW` = 1 will centralize the Schur complement matrix, *stored by columns* (instead of by rows as in the `ICNTL(19)`=1 option). More details on this are presented in Subsection 5.14.3.

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

In order to have a centralized Schur complement matrix by columns, it is possible to use a particular case of the distributed Schur complement (`ICNTL(19)`=2 or 3, see Subsection 5.14.2), where the Schur complement is only assigned to one processor (`NPCOL` × `NPROW` = 1). Therefore we refer the reader to the previous section for a detailed description of the parameters for using this option. This option is recommended compared to `ICNTL(19)`=1 (centralized Schur complement by rows).

The Schur complement matrix will be available on the host node if `PAR`=1, and on the node with MPI identifier 1 (first working slave processor) if `PAR`=0.

Let us summarize 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 host[8]:

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 from the factorization phase, the pointer array `SCHUR` available on the host contains the Schur complement. If the matrix is unsymmetric (`SYM`=0), then the settings `ICNTL(19)`=2 and `ICNTL(19)`=3 have an identical behaviour and the unsymmetric Schur complement is returned by columns (i.e., in column-major format). If the matrix is symmetric (`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

---

[8]As 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.

is returned by rows and the lower triangular part is not accessed. If the matrix is symmetric (`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.

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

As explained in Subsection 3.16, when a Schur complement has been computed during the factorization phase, either the solution phase computes a solution on the internal problem (`ICNTL(26)`=0) or the complete problem can be used to first condensed the right-hand side on the Schur variables (`ICNTL(26)`=1). Then the condensed right-hand side is made available to the user for computing the local solution using the Schur matrix. This local solution corresponding to Schur variables can then be expanded to compute a global solution (`ICNTL(26)`=2).

ICNTL(26) drives the solution phase if a Schur complement matrix has been computed (`ICNTL(19)` $\neq$ 0, see Subsection 3.16 for details)

>   *Phase*: accessed by the host during the solution phase. It will be accessed also during factorization if the forward elimination is performed during factorization (`ICNTL(32)`=1)

>   *Possible variables/arrays involved*: `REDRHS`, `LREDRHS`

>   *Possible values* :

>   0 : standard solution phase on the internal problem; referring to the notations from Subsection 3.16, only the system $\mathbf{A}_{1,1}x_1 = b_1$ is solved and the entries of the right-hand side corresponding to the Schur are explicitly set to 0 on output.

>   1 : condense/reduce the right-hand side on the Schur. 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 the pointer array `REDRHS`, that must be allocated by the user. Its leading dimension `LREDRHS` must be provide, too.

>   2 : expand the Schur local solution on the complete solution variables. `REDRHS` is considered to be the solution corresponding to the Schur variables. It must be allocated by the user as well as its leading dimension `LREDRHS` must be provided. 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.

>   *Default value*: 0 (normal solution phase)

>   *Incompatibility*: if ICNTL(26) = 1 or 2, then error analysis and iterative refinement are disabled (`ICNTL(11)` and `ICNTL(10)`)

>   *Related parameters*: `ICNTL(19)`, `ICNTL(32)`

>   *Remarks*: If ICNTL(26) $\neq$ 0, then the user should provide workspace in the pointer array `REDRHS`, as well as a leading dimension `LREDRHS` (see Subsection 5.14). Note that if no Schur complement was computed, ICNTL(26) = 1 or 2 results in an error.

If the complete system should be solved using the Schur complement matrix (see Subsection 3.16, `ICNTL(26)`=1 or 2), then the following parameters must be defined on the host on entry to the solution step:

The right-hand side matrix $[B_1 B_2]^T$ (see Figure 3) must be defined on input of the solve phase. The user can input the right-hand side matrix as a dense matrix (`ICNTL(20)` = 0, RHS, `LRHS`, `NRHS`, see Subsection 5.13.1) or as a sparse matrix (`ICNTL(20)` = 1, `NZ_RHS`, `NRHS`, `RHS_SPARSE`, `IRHS_SPARSE`, `IRHS_PTR`, see Subsection 5.13.2).

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 and is greater than 1. In that case, it must be larger or equal to `SIZE_SCHUR`, the size of the Schur complement. If `NRHS` is not provided (or is equal to 1), `LREDRHS` needs not be provided.

$$\begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & S \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

Figure 3: Solving the complete system using the Schur complement matrix

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 the reduction/condensation phase should be performed (`ICNTL(26)`=1), then on exit from the solution phase, REDRHS(i+(k-1)*`LREDRHS`), i=1, ..., `SIZE_SCHUR`, k=1, ..., `NRHS` will hold the reduced right-hand side (the y_2 vector of Equation (13)).

If the expansion phase should be performed (`ICNTL(26)`=2), then REDRHS(i+(k-1)*`LREDRHS`), i=1, ..., `SIZE_SCHUR`, k=1, ..., `NRHS` must be set (on entry to the solution phase) to the solution on the Schur variables (the x_2 vector of Equation (14)). In this case (i.e., `ICNTL(26)`=2) `REDRHS` is not altered by `MUMPS`.

Note that on exit, the solution matrix $[X_1 X_2]^T$ in Figure 3 is stored in the RHS parameter, except in case of distributed solution where it will be stored in `ISOL_loc` and `SOL_loc` (see `ICNTL(21)` and Subsection 5.13).

5.15 Block Low Rank (BLR) factorization (`ICNTL(35) and CNTL(7)`)

The current version of `MUMPS` implements the low-rank factorization strategy defined in [47, 3] (see also Subsection 3.17): factor matrices are represented in Block Low Rank (BLR) format in order to reduce the amount of operations performed during factorization. Blocks are compressed using a truncated Rank-Revealing QR factorization (RRQR). This preliminary version does not include the use of compressed factor matrices during the solution phase. Note that the panels (see Figure 1) are factorized with a full-rank algorithm, making BLR compatible with numerical pivoting.

In this section, we first describe how to enable this BLR functionality in a multithreaded environment (Subsection 5.15.1), we then describe the preliminary Application Program Interface (API) to control the use of the Block Low Rank factorization (Subsection 5.15.2), and finally comment on the statistics that are provided on output (Subsection 5.15.3).

5.15.1 Enabling the BLR functionality at installation

It is strongly advised to interface MUMPS with external ordering packages (Metis [34] or SCOTCH [39]) to enable the BLR representation of frontal matrix to rely on K-way partitionning [47].

On multiprocessor nodes with a shared memory it is strongly advised to use multithreading and combine multithreaded BLAS with OpenMP directives-based multithreading. For the GNU fortran compiler `-fopenmp` should thus be added to `OPTF` and `OPTL` in `Makefile.inc`. For the Intel compiler `-openmp` should be added. `OMP_NUM_THREADS` should then be set at run time.

In case you use the multithreaded implementation of the BLAS routines provided by Intel in the MKL library, we also recommend adding the option `-DBLR_MT`. This will allow for using both multithreaded BLAS and sequential BLAS in appropriate portions of the code (serial BLAS when parallelism comes from OpenMP parallel regions and multithreaded BLAS when there is not enough parallelism from OpenMP), with a significant impact on performance.

If the number of threads from your BLAS implementation cannot vary at execution time and should be constant in all BLAS calls, then we recommend to either use a multithreaded BLAS library but without `-DBLR_MT`, or, if you do not have a multithreaded BLAS library available, to use `-DBLR_MT` with a serial

BLAS library. As these two options will lead to suboptimal performance, we recommend to use MKL when it is available.

To summarize, in order to have MUMPS fully functional for BLR, one should set the compiler options in `Makefile.inc` as follows:

- GNU compiler options to be added to `OPTF` and `OPTL`: `-fopenmp`
- INTEL compiler options to be added to `OPTF` and `OPTL`: `-openmp`
- If you use multithreaded BLAS from the MKL library (or a multithreaded BLAS library compatible with OpenMP that uses the number of available OpenMP threads), or a sequential BLAS library, add to `OPTF`: `-DBLR_MT`

For other compilers, please replace the `-fopenmp`/`-openmp` flag with the corresponding flag for OpenMP compilation.

Please note that, low-rank approximations are computed using a truncated Rank-Revealing QR (RRQR) factorization implemented as a variant of the LAPACK `_GEQP3` and `_LAQPS` routines; linking with a LAPACK library is necessary to satisfy the dependencies of this feature. On many systems, BLAS and LAPACK routines are provided by a single library (e.g., the Intel MKL library) but, if it's not the case, the LAPACK library can simply be added in the `LAPACK` variable of the MUMPS `Makefile.inc` file.

5.15.2 Application Program Interface

In order to activate the BLR feature and define the dropping parameter ε used for the approximations, the following parameters have been introduced.

ICNTL(35) controls the activation of the BLR feature

> *Phase*: accessed by the host during the analysis phase
>
> *Possible values* :
>
> 0 : Standard multifrontal factorization (BLR is not activated)
>
> 1 : BLR approximation is activated (precision contoled by `CNTL(7)`).
>
> Other values are treated as 0.
>
> *Default value*: 0 (standard multifrontal factorization)
>
> *Related parameters*: `CNTL(7)`
>
> *Incompatibility*: For the number of operations performed when BLR is activated see `RINFOG(14)`. Note that the activation of BLR feature is currently incompatible with elemental matrices (`ICNTL(5)` = 1) and with forward elimination during factorization (`ICNTL(32)` = 1).

CNTL(7) is the dropping parameter (ε) (double precision real value) controlling the accuracy of the low-rank approximation. This value is used as a stopping criterion for the compression of BLR blocks which is achieved through a truncated Rank Revealing QR factorization. More precisely, to compute the low-rank form of a block, we perform a **QR** factorization with column pivoting which is stopped as soon as a diagonal coefficient of the R factor falls below the threshold, i.e., when $\|r_{kk}\| < \varepsilon$. This is implemented as a variant of the LAPACK [20] `_GEQP3` routine. Larger values of this parameter lead to more compression at the price of a lower accuracy. Note that ϵ is used as an **absolute** tolerance, i.e., not relative to the input matrix, or the frontal matrix or the block norms; for this reason we recommend to let the solver automatically preprocess (e.g., scale) the input matrix.

> *Phase*: accessed by the host during the factorization phase when `ICNTL(35)` = 1
>
> *Possible values* :
>
> 0.0 : full precision approximation.
>
> > 0.0 : the dropping parameter is **CNTL(7)**.
>
> *Default value*: 0.0 (full precision (i.e., no approximation)).
>
> *Related parameters*: `ICNTL(35)`
>
> *Remarks*: Note that, depending on the application, gains can be expected even with small values (close to machine precision) of **CNTL(7)**.

5.15.3   BLR output: statistics

The output below gives execution statistics related to the BLR feature described in Section 3.17.

```
-------------- Beginning of BLR statistics --------------------------------

----------------------------------------------
Settings for Block Low-Rank (BLR) are :
----------------------------------------------
BLR ALGORITHM CHARACTERISTICS
    Variant used:  FSCU
    Target BLR block size (variable) = 128 - 512
    RRQR precision (epsilon) = 1.0E-06


-------------------------------------------
Statistics after BLR factorization
-------------------------------------------
    Number of BLR fronts = 847

STATISTICS ON OPERATION COUNT (OPC)
    Total theoretical full-rank OPC (i.e.  FR OPC) = 2.572E+13 (100.0%)
    Total effective OPC (% FR OPC) = 1.482E+12 ( 5.8%)

-------------- End of BLR statistics --------------------------------------
```

This output reports the following information:

- `Variant used`: as explained in Section 3.17, several variants of the BLR method have been implemented. However, in this preliminary version of the feature, only the `FSCU` one is provided.

- `Target BLR block size`: because MUMPS uses a variable block size, this actually reports the minimum and maximum BLR block sizes. This range is fixed to values that were found to perform well on a wide range of problems.

- `RRQR precision`: this is the actual value of the threshold used for computing the low-rank approximation of blocks, which is set through the **CNTL(7)** control parameter.

- `Number of BLR fronts`: not all fronts are compressed using low-rank approximation but only those that fulfill a minimum size requirement (set to a value that was found to perform well on a wide range of problems).

- `Total theoretical full-rank OPC`: this is the number of floating-point operations required to achieve the full-rank factorization. It is provided here as a reference.

- `Total effective OPC`: this is the actual number of floating-point operations done by the factorization phase with BLR feature. The value reported in parentheses is the fraction of full-rank operation count.

Using the BLR feature, the reduction in time for factorization should be related to the percentage reduction of full rank operations.

6   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.

6.1   Integer control parameters

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

- ICNTL(1) is the output stream for error messages
- ICNTL(2) is the output stream for diagnostic printing, statistics, and warning message
- ICNTL(3) is the output stream for global information, collected on the host
- ICNTL(4) is the level of printing for error, warning, and diagnostic messages
- ICNTL(5) controls the matrix input format
- ICNTL(6) permutes the matrix to a zero-free diagonal and/or scale the matrix
- ICNTL(7) computes a symmetric permutation in case of sequential analysis
- ICNTL(8) describes the scaling strategy
- ICNTL(9) computes the solution using $\mathbf{A}$ or $\mathbf{A}^T$
- ICNTL(10) applies the iterative refinement to the computed solution
- ICNTL(11) computes statistics related to an error analysis
- ICNTL(12) defines an ordering strategy for symmetric matrices
- ICNTL(13) controls the parallelism of the root node
- ICNTL(14) controls the percentage increase in the estimated working space
- ICNTL(18) defines the strategy for the distributed input matrix
- ICNTL(19) computes the Schur complement matrix
- ICNTL(20) determines the format (dense or sparse) of the right-hand sides
- ICNTL(21) determines the distribution (centralized or distributed) of the solution vectors
- ICNTL(22) controls the in-core/out-of-core (OOC) factorization and solve
- ICNTL(23) corresponds to the maximum size of the working memory in MegaBytes that MUMPS can allocate per working processor
- ICNTL(24) controls the detection of "null pivot rows"
- ICNTL(25) allows the computation of a solution of a deficient matrix and also of a null space basis
- ICNTL(26) drives the solution phase if a Schur complement matrix
- ICNTL(27) controls the blocking size for multiple right-hand sides
- ICNTL(28) determines whether a sequential or parallel computation of the ordering is performed
- ICNTL(29) defines the parallel ordering tool to be used to compute the fill-in reducing permutation
- ICNTL(30) computes a user-specified set of entries in the inverse $\mathbf{A}^{-1}$ of the original matrix
- ICNTL(31) indicates which factors may be discarded during the factorization
- ICNTL(32) performs the forward elimination of the right-hand sides during the factorization
- ICNTL(33) computes the determinant of the input matrix
- ICNTL(35) controls the activation of Block Lock Rank (BLR) based factorization

ICNTL(1) is the output stream for error messages.

Possible values :

≤ 0: these messages will be suppressed.

> 0 : is the output stream.

Default value: 6 (standard output stream)

ICNTL(2) is the output stream for diagnostic printing, statistics, and warning messages.

Possible values :

≤ 0: these messages will be suppressed.

> 0 : is the output stream.

Default value: 0

ICNTL(3) is the output stream for global information, collected on the host.

Possible values :

≤ 0: these messages will be suppressed.

> 0 : is the output stream.

Default value: 6 (standard output stream)

ICNTL(4) is the level of printing for error, warning, and diagnostic messages.

Possible values :

≤ 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, warnings and information on input, output parameters printed.

Default value: 2 (errors and warnings printed)

ICNTL(5) controls the matrix input format (see Subsection 5.2.2).

Phase: accessed by the host and only during the analysis phase

Possible variables/arrays involved: N, NNZ (or NZ for backward compatibility), IRN, JCN, NNZ_loc (or NZ_loc for backward compatibility), IRN_loc, JCN_loc, A_loc, NELT, ELTPTR, ELTVAR, and A_ELT

Possible values :

0 : assembled format. The matrix must be input in the structure components N, NNZ (or NZ), IRN, JCN, and A if the matrix is centralized on the host (see Subsection 5.2.2.1) or in the structure components N, NNZ_loc (or NZ_loc)), IRN_loc, JCN_loc, A_loc if the matrix is distributed (see Subsection 5.2.2.2).

1 : elemental format. The matrix must be input in the structure components N, NELT, ELTPTR, ELTVAR, and A_ELT (see Subsection 5.2.2.3).

Any other values will be treated as 0.

Default value: 0 (assembled format)

Related parameters: `ICNTL(18)`

Remarks: NNZ and NNZ_loc are 64-bit integers (NZ and NZ_loc are 32-bit integers kept for backward compatibility and will be obsolete in future releases).

Parallel analysis (`ICNTL(28)` =2) is only available for matrices in assembled format and, thus, an error will be raised for elemental matrices (ICNTL(5)=1).

Elemental matrices can be input only centralized on the host (`ICNTL(18)` =0).

ICNTL(6) permutes the matrix to a zero-free diagonal and/or scale the matrix (see Subsection 3.2 and Subsection 5.3.2).

Phase: accessed by the host and only during sequential analysis (`ICNTL(28)`=1)

Possible variables/arrays involved: optionally `UNS_PERM`, mumps_par%A, `COLSCA` and `ROWSCA`

Possible values :

0 : No column permutation is computed.

1 : The permuted matrix has as many entries on its diagonal as 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. The numerical values of the original matrix, (mumps_par%A), must be provided by the user during the analysis phase.

3 : Variant of option 2 with different performance. The numerical values of the original matrix (mumps_par%A) must be provided by the user during the analysis phase.

4 : The sum of the diagonal entries of the permuted matrix is maximized. The numerical values of the original matrix (mumps_par%A) must be provided by the user during the analysis phase.

5 : The product of the diagonal entries of the permuted matrix is maximized. Scaling vectors are also computed and stored in `COLSCA` and `ROWSCA`, if ICNTL(8) is set to -2 or 77. With these scaling vectors, 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. For unsymmetric matrices, `COLSCA` and `ROWSCA` are meaningful on the permuted matrix $\mathbf{A} \ \mathbf{Q_c}$ (see Equation (5)). For symmetric matrices, `COLSCA` and `ROWSCA` are meaningful on the original matrix $\mathbf{A}$. The numerical values of the original matrix, mumps_par%A, must be provided by the user during the analysis phase.

6 : Similar to 5 but with a different algorithm. The numerical values of the original matrix, mumps_par%A, must be provided by the user during the analysis phase.

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

Default value: 7 (automatic choice done by the package)

Incompatibility: If the matrix is symmetric positive definite ($SYM = 1$), or in elemental format (`ICNTL(5)`=1), or the parallel analysis is requested (`ICNTL(28)`=2) 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)`=1,2,3), then ICNTL(6) is treated as 0.

Related parameters: `ICNTL(8)`, `ICNTL(12)`

Remarks: *On assembled centralized unsymmetric matrices* (`ICNTL(5)`=0, `ICNTL(18)`=0, SYM = 0), if ICNTL(6)=1, 2, 3, 4, 5, 6 a column permutation (based on weighted bipartite matching algorithms described in [25, 26]) is applied to the original matrix to get a zero-free diagonal. The user is advised to set ICNTL(6) to a nonzero value when the matrix is very unsymmetric in structure. On output to the analysis phase, when the column permutation is not the identity, the pointer `UNS_PERM` (internal data valid until a call to `MUMPS` with `JOB`=-2) provides access to the permutation on the host processor (see Subsection 5.3.1). Otherwise, the pointer is not associated. The column permutation is such that entry $a_{i,perm(i)}$ is on the diagonal of the permuted matrix.

On general assembled centralized symmetric matrices (`ICNTL(5)`=0, `ICNTL(18)`=0, SYM = 2), if ICNTL(6)=1, 2, 3, 4, 5, 6, the column permutation is internally used to determine a set of recommended 1×1 and 2×2 pivots (see [27] and the description of `ICNTL(12)` in Subsection 6.1 for more details). 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 `UNS_PERM` is not associated.

ICNTL(7) computes a symmetric permutation (ordering) to determine the pivot order to be used for the factorization in case of sequential analysis (`ICNTL(28)`=1). See Subsection 3.2 and Subsection 5.4.

Phase: accessed by the host and only during the *sequential* analysis phase (`ICNTL(28) = 1`).

Possible variables/arrays involved: `PERM_IN`, `SYM_PERM`

Possible values :

- 0 : Approximate Minimum Degree (AMD) [8] 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, ... N) must hold the position of variable i in the pivot order. In other words, row/column i in the original matrix corresponds to row/column `PERM_IN`(i) in the reordered matrix.
- 2 : Approximate Minimum Fill (AMF) is used,
- 3 : SCOTCH[9] [39] package is used if previously installed by the user otherwise treated as 7.
- 4 : PORD[10] [43] is used if previously installed by the user otherwise treated as 7.
- 5 : the Metis[11] [34] package is used if previously installed by the user otherwise treated as 7.
- 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.

Default value: 7 (automatic choice)

Incompatibility: ICNTL(7) is meaningless if the parallel analysis is chosen (`ICNTL(28)`=2).

Related parameters: `ICNTL(28)`

Remarks: Even when the ordering is provided by the user, the analysis must be performed before numerical factorization.

For *assembled matrices (centralized or distributed)* (`ICNTL(5)`=0) all the options are available.

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.

If the user asks for *a Schur complement matrix* (`ICNTL(19)` = 1, 2, 3) and

- the matrix is assembled (`ICNTL(5)`=0) then only options 0, 1, 5 and 7 are currently available. Other options are treated as 7.
- the matrix is elemental (`ICNTL(5)`=1) only options 0, 1 and 7 are currently available. Other options are treated as 7 which will (currently) be treated as 0 (AMD).
- in both cases (assembled or elemental matrix) if the pivot order is given by the user (ICNTL(7)=1) then the following property should hold: `PERM_IN`(`LISTVAR_SCHUR`(i)) = `N-SIZE_SCHUR`+i, for i=1,`SIZE_SCHUR`.

For matrices with *relatively dense rows*, we highly recommend option 6 which may significantly reduce the time for analysis.

On output, the pointer array `SYM_PERM` provides access, on the host processor, to the symmetric permutation that is effectively computed during the analysis phase by the MUMPS package, and `INFOG(7)` to the ordering option that was effectively chosen. In fact, the option corresponding to ICNTL(7) may be forced by MUMPS when for example the ordering option chosen by the user is not compatible with the value of `ICNTL(12)` or the necessary package is not installed.

`SYM_PERM`(i), i=1, ... N, holds the position of variable i in the pivot order. In other words, row/column i in the original matrix corresponds to row/column `SYM_PERM`(i) in the reordered matrix. See also Subsection 5.4.1.

---

[9]See http://gforge.inria.fr/projects/scotch/ to obtain a copy.
[10]Distributed within MUMPS by permission of J. Schulze (University of Paderborn).
[11]See http://glaros.dtc.umn.edu/gkhome/metis/metis/overview to obtain a copy.

ICNTL(8) describes the scaling strategy (see Subsection 5.3).

> *Phase*: accessed by the host during analysis phase (that need be sequential `ICNTL(28)`=1) or on entry to numerical factorization phase
>
> *Possible variables/arrays involved*: `COLSCA`,`ROWSCA`
>
> *Possible values* :
>
> -2: Scaling computed during analysis (see [25, 26] for the unsymmetric case and [27] for the symmetric case). The user has to provide the numerical values of the original matrix (mumps_par%A) on entry to the analysis.
>
> -1: Scaling provided by the user. Scaling arrays must be provided in `COLSCA` and `ROWSCA` on entry to the numerical factorization phase by the user, who is then responsible for allocating and freeing them. If the input matrix is symmetric (`SYM`= 1 or 2), then the user should ensure that the array `ROWSCA` is equal to (or points to the same location as) the array `COLSCA`.
>
> 0 : No scaling applied/computed.
>
> 1 : Diagonal scaling 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,
>
> 7 : Simultaneous row and column iterative scaling based on [42] and [17] computed during the numerical factorization phase.
>
> 8 : Similar to 7 but more rigorous and expensive to compute; computed during the numerical factorization phase.
>
> 77 : Automatic choice of the value of ICNTL(8) done during analysis.
>
> *Default value*: 77 (automatic choice done by the package)
>
> *Related parameters*: `ICNTL(6)`, `ICNTL(12)`
>
> *Remarks*: If 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 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 phase. Note that scalings can be efficiently computed during analysis when requested (see `ICNTL(6)` and `ICNTL(12)`).
>
> If the input matrix is symmetric (`SYM`= 1 or 2), then only options –2, –1, 0, 1, 7, 8 and 77 are allowed and other options are treated as 0.
>
> If the input matrix is in elemental format (`ICNTL(5)` = 1), then only options –1 and 0 are allowed and other options are treated as 0.
>
> If the initial assembled matrix is distributed (`ICNTL(18)`=1,2,3 and `ICNTL(5)` = 0), then only options 7, 8 and 77 are allowed, otherwise no scaling is applied.

ICNTL(9) computes the solution using $\mathbf{A}$ or $\mathbf{A}^T$

> *Phase*: accessed by the host during the solve phase.
>
> *Possible values* :
>
> 1 : $\mathbf{AX} = \mathbf{B}$ is solved.
>
> $\neq 1$: $\mathbf{A}^T\mathbf{X} = \mathbf{B}$ is solved.
>
> *Default value*: 1
>
> *Related parameters*: `ICNTL(10)`, `ICNTL(11)`, `ICNTL(21)`, `ICNTL(32)`
>
> *Remarks*: when a forward elimination is performed during the factorization (see `ICNTL(32)`) only `ICNTL(9)`=1 is allowed.

ICNTL(10) applies the iterative refinement to the computed solution (see Subsection 5.5).

> *Phase*: accessed by the host during the solve phase.

Possible variables/arrays involved: NRHS

Possible values :

< 0 : Fixed number of steps of iterative refinement. No stopping criterion is used.

  0 : No iterative refinement.

> 0 : Maximum number of steps of iterative refinement. A stopping criterion is used, therefore a test for convergence is done at each step of the iterative refinement algorithm.

Default value: 0 (no iterative refinement)

Related parameters: CNTL(2)

Incompatibility: if ICNTL(21)=1 (solution kept distributed) or if ICNTL(32)=1 (forward elimination during factorization), or if NRHS>1 (multiple right hand sides), then ICNTL(10) is treated as 0.

Remarks: Note that if ICNTL(10) < 0, |ICNTL(10)| steps of iterative refinement are performed, without any test of convergence (see Algorithm 3). This means that the iterative refinement may diverge, that is the solution instead of being improved may be worse from an accuracy point of view. But it has been shown [21] that with only two to three steps of iterative refinement the solution can often be significantly improved. So if the convergence test should not be done we recommend to set ICNTL(10) to -2 or -3.

Note also that it is not necessary to activate the error analysis option (ICNTL(11)= 1,2) to be able to run the iterative refinement with stopping criterium (ICNTL(10) > 0). However, since the backward errors ω_1 and ω_2 have been computed, they are still returned in RINFOG(7) and RINFOG(8), respectively.

It must be noticed that iterative refinement with stopping criterium (ICNTL(10) > 0) will stop when

1. either the requested accuracy is reached ($\omega_1 + \omega_2 <$ CNTL(2))
2. or when the convergence rate is too slow ($\omega_1 + \omega_2$ does not decrease by at least a factor of 5)
3. or when exactly ICNTL(10) steps have been performed.

In the first two cases the number of iterative refinement steps (INFOG(15)) may be lower than ICNTL(10).

ICNTL(11) computes statistics related to an error analysis of the linear system solved ($\mathbf{Ax} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ (see ICNTL(9))). See Subsection 5.6.

Phase: accessed by the host and only during the solve phase.

Possible variables/arrays involved: NRHS

Possible values :

  0 : no error analysis is performed (no statistics).

  1 : compute all the statistics (very expensive).

  2 : compute main statistics (norms, residuals, componentwise backward errors), but not the most expensive ones like (condition number and forward error estimates).

Values different from 0, 1, and 2 are treated as 0.

Default value: 0 (no statistics).

Incompatibility: if NRHS > 1, if ICNTL(32)=1 (forward elimination during factorization), or if ICNTL(21)=1 (solution kept distributed) then error analysis is not performed and ICNTL(11) is treated as 0

Related parameters: ICNTL(9)

Remarks: The computed statistics are returned in various informational parameters, see also Subsection 3.3:

– If ICNTL(11)= 2, then the infinite norm of the input matrix ($\|A\|_\infty$ or $\|A^T\|_\infty$ in RINFOG(4)), the infinite norm of the computed solution ($\|\bar{x}\|_\infty$ in RINFOG(5)), and the scaled residual $\frac{\|A\bar{x}-b\|_\infty}{\|A\|_\infty \|\bar{x}\|_\infty}$ in RINFOG(6), a componentwise backward error estimate in RINFOG(7) and RINFOG(8) are computed.

– If ICNTL(11)= 1, then in addition to the above statistics also an estimate for the error in the solution in RINFOG(9), and condition numbers for the linear system in RINFOG(10) and RINFOG(11) are also returned.

If performance is critical, ICNTL(11) should be set to 0. If both performance is critical and statistics are requested, then ICNTL(11) should be set to 2. If ICNTL(11)=1, the error analysis is very costly (typically significantly more costly than the solve phase itself).

ICNTL(12) defines an *ordering strategy for symmetric matrices* (SYM = 2) (see [27] for more details) and is used, in conjunction with ICNTL(6), to add constraints to the ordering algorithm (ICNTL(7) option).

Phase: accessed by the host and only during the analysis phase.

Possible values :

  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.

Default value: 0 (automatic choice).

Incompatibility: If the matrix is unsymmetric (SYM=0) or symmetric definite positive matrices (SYM=1), or the matrix is in elemental format (ICNTL(5)=1), or the matrix is initially distributed (ICNTL(18)=1,2,3) or the ordering is provided by the user (ICNTL(7)=1), or the Schur option (ICNTL(19) $\neq$ 0) is required, ICNTL(12) is treated as 1 (nothing done).

Related parameters: ICNTL(6), ICNTL(7)

Remarks: 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 (scaling computed during analysis).

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) controls the parallelism of the root node (enabling or not the use of ScaLAPACK) and also its splitting.

Phase: accessed by the host during the analysis phase.

Possible values :

$\leq$ -1 : treated as 0.

  -1 : force splitting of the root node in all cases (even sequentially)

  0 : parallel factorization of the root node. If the size of the root frontal node (last Schur complement to be factored) is larger than an internal threshold, then ScaLAPACK will be used for factorizing it. Otherwise, the root node will be processed by a single MPI process.

  > 0 : forces a sequential factorization of the root node (ScaLAPACK will not be used). In this case if the number of working processors is strictly larger than ICNTL(13) 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 (advised value is 1).

Default value: 0 (parallel factorization on the root node)

Remarks: Processing the root sequentially (ICNTL(13) > 0) 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 Subsection 5.9).

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) controls the percentage increase in the estimated working space, see Subsection 5.8.

Phase: accessed by the host both during the analysis and the factorization phases.

Default value: 20 (which corresponds to a 20 % increase).

Related parameters: `ICNTL(23)`

Remarks: When significant extra fill-in is caused by numerical pivoting, increasing ICNTL(14) may help.

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

ICNTL(18) defines the strategy for the distributed input matrix (only for assembled matrix, see Subsection 5.2.2).

Phase: accessed by the host during the analysis phase.

Possible values :

  0 : the input matrix is centralized on the host (see Subsection 5.2.2.1).
  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 entries distributed according to the mapping on entry to the numerical factorization phase (see Subsection 5.2.2.2).
  2 : the user provides the structure of the matrix on the host at analysis, and the distributed matrix entries on all slave processors at factorization. Any distribution is allowed (see Subsection 5.2.2.2).
  3 : user directly provides the distributed matrix, pattern and entries, input both for analysis and factorization (see Subsection 5.2.2.2).

Other values are treated as 0.

Default value: 0 (input matrix centralized on the host)

Related parameters: `ICNTL(5)`

Remarks: In case of distributed matrix, we recommand options 2 or 3. Among them, we recommand option 3 which is easier to use. Option 1 is kept for backward compatibility but is deprecated and we plan to suppress it in a future release.

ICNTL(19) computes the Schur complement matrix (see Subsection 5.14).

Phase: accessed by the host during the analysis phase.

Possible variables/arrays involved: `SIZE_SCHUR`, `LISTVAR_SCHUR`, `NPROW`, `NPCOL`, `MBLOCK`, `NBLOCK`, `SCHUR`, `SCHUR_MLOC`, `SCHUR_NLOC`, and `SCHUR_LLD`

Possible values :

  0 : complete factorization. No Schur complement is returned.
  1 : the Schur complement matrix will be returned centralized by rows on the host after the factorization phase. On the host before the analysis phase, the user must set the integer variable `SIZE_SCHUR` to the size of the Schur matrix, the integer pointer array `LISTVAR_SCHUR` to the list of indices of the Schur matrix.
  2 or 3 : the Schur complement matrix will be returned distributed by columns: the Schur will be returned on the slave processors in the form of a 2D block cyclic distributed matrix (ScaLAPACK style) after factorization. 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 Subsection 5.14). On the host before the

analysis phase, the user must set the integer variable SIZE_SCHUR to the size of the Schur matrix, the integer pointer array LISTVAR_SCHUR to the list of indices of the Schur matrix. The integer variables NPROW, NPCOL, MBLOCK, NBLOCK may also be defined (default values will otherwise be provided).

Values not equal to 1, 2 or 3 are treated as 0.

Default value: 0 (complete factorization)

Incompatibility: 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.

Related parameters: ICNTL(7)

Remarks: If the ordering is given (ICNTL(7)=1) then the following property should hold: PERM_IN(LISTVAR_SCHUR(i)) = N-SIZE_SCHUR+i, for i=1,SIZE_SCHUR.

Note that, in order to have a centralized Schur complement matrix by columns (see Subsection 5.14.3), it is possible (and recommended) to use a particular case of the distributed Schur complement (ICNTL(19)=2 or 3), where the Schur complement is assigned to only one processor (NPCOL × NPROW = 1).

ICNTL(20) determines the format (dense or sparse) of the right-hand sides

Phase: accessed by the host during the solve phase.

Possible variables/arrays involved: RHS, NRHS, LRHS, IRHS_SPARSE, RHS_SPARSE, IRHS_PTR and NZ_RHS.

Possible values :

  0 : the right-hand side is in dense format in the structure component RHS, NRHS, LRHS (see Subsection 5.13.1)

 1,2,3 : the right-hand side is in sparse format in the structure components IRHS_SPARSE, RHS_SPARSE, IRHS_PTR and NZ_RHS.

   1 : The decision of exploiting sparsity of the right-hand side to accelerate the solution phase is done automatically.
   2 : Sparsity of the right-hand side is NOT exploited to improve solution phase.
   3 : Sparsity of the right-hand side is exploited during solution phase.

Values different from 0, 1, 2 or 3 are treated as 0. For a sparse right-hand side, the recommended value is 1.

Default value: 0 (dense right-hand sides)

Incompatibility: When ICNTL(20)=0 (dense right-hand side) and NRHS > 1 (multiple right-hand side) the functionalities related to iterative refinement (ICNTL(10)) and error analysis (ICNTL(11)) are currently disabled.

With sparse right-hand sides (ICNTL(20)=1,2,3), the forward elimination during the factorization (ICNTL(32)=1) is not currently available.

Remarks: For details on how to set the input parameters see Subsection 5.13.1 and Subsection 5.13.2. Please note that duplicate entries in the sparse right-hand sides are summed.

ICNTL(21) determines the distribution (centralized or distributed) of the solution vectors.

Phase: accessed by the host during the solve phase.

Possible variables/arrays involved: RHS, ISOL_loc and SOL_loc, LSOL_loc

Possible values :

  0 : the solution vector is assembled and stored in the structure component RHS (gather phase), that must have been allocated earlier by the user (see Subsection 5.13.4).

1 : the solution vector is kept distributed 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 (see Subsection 5.13.5).

Values different from 0 and 1 are currently treated as 0.

Default value: 0 (assembled centralized format)

Incompatibility: If the solution is kept distributed, error analysis and iterative refinement (controlled by `ICNTL(10)` and `ICNTL(11)`) are not applied.

ICNTL(22) controls the in-core/out-of-core (OOC) factorization and solve.

Phase: accessed by the host during the factorization phase.

Possible variables/arrays involved: `OOC_TMPDIR` and `OOC_PREFIX`

Possible values :

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 Subsection 3.13).

Default value: 0 (in-core factorization)

Remarks: The variables `OOC_TMPDIR` and `OOC_PREFIX` are used to indicate the directory and the prefix, respectively, where to store the factors. They must be set after the initialization phase (`JOB` = -1) and before the factorization phase (`JOB`=2,4,5 or 6). Otherwise, `MUMPS` will use the `/tmp` directory and arbitrary file names. Note `MUMPS` accesses to the variables `OOC_TMPDIR` and `OOC_PREFIX` only during the factorization phase. Several files under the same directory and with the same prefix are created to store the factors. Their names contain a unique hash and `MUMPS` is in charge of keeping trace of them.

The files containing the factors will be deleted if a new factorization starts or when a termination phase (`JOB`=-2) is called.

Note that, in case of abnormal termination of an application calling `MUMPS` (for example, a termination of the calling process with a segmentation fault, or, more generally, a termination of the calling process without a call to `MUMPS` with `JOB`=-2), the files containing the factors are not deleted. It is then the user's responsibility to delete them, as shown in bold in the example below, where the application calling `MUMPS` is launched from a bash script and environment variables are used to define the OOC environment:

```
#!/bin/bash
export MUMPS_OOC_TMPDIR="/local/mumps_data/"
export MUMPS_OOC_PREFIX="job_myapp_"
mpirun -np 128 ./myapplication
# Suppress MUMPS OOC files in case of bad application termination
rm -f ${MUMPS_OOC_TMPDIR}/${MUMPS_OOC_PREFIX}*
```

ICNTL(23) corresponds to the maximum size of the working memory in MegaBytes that `MUMPS` can allocate per working processor, see Subsection 5.8 for more details.

Phase: accessed by the host at the beginning of the factorization phase and is only significant on the host.

Possible values :

0 : each processor will allocate workspace based on the estimates computed during the analysis

>0 : maximum size of the working memory in MegaBytes per working processor to be allocated

Default value: 0

Related parameters: `ICNTL(14)`

Remarks: 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 for the factorization phase a workspace based on the estimates computed during the analysis (`INFO(17)` that is the estimate of the sum over all processors) 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 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 the maximum over all processors: `INFOG(16)` in the in-core case, or `INFOG(26)` in the out-of-core case.

ICNTL(24) controls the detection of "null pivot rows".

Phase: accessed by the host during the factorization phase

Possible variables/arrays involved: `PIVNUL_LIST`

Possible values :

  0: Nothing done. A null pivot will result in error `INFO(1)`=--10.

  1: Null pivot row/column detection.

Other values are treated as 0.

Default value: 0 (no null pivot detection)

Related parameters: `CNTL(3)`, `CNTL(5)`, `ICNTL(13)`, `ICNTL(25)`

Remarks:

`CNTL(3)` is used to compute the threshold to decide if a pivot row is "null".

Null pivot rows are modified to enable the solution phase to provide one solution among the possible solutions of the numerically deficient matrix. The parameter `CNTL(5)` defines the fixation of null pivots.

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, 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)`).

Note that when ScaLAPACK is applied on the root node (see `ICNTL(13)` = 0), then exact null pivots on the root will stop the factorization (`INFO(1)`=--10) while if tiny pivots are present on the root node the ScaLAPACK routine will factorize the root matrix. Computing the root node factorization sequentially (this can be forced by setting `ICNTL(13)` to 1) will help with the correct detection of null pivots but may degrade performance.

ICNTL(25) allows the computation of a solution of a deficient matrix and also of a null space basis.

Phase: accessed by the host during the solution phase

Possible variables/arrays involved: RHS, `ISOL_loc` and `SOL_loc`

Possible values :

  0: A normal solution step is performed. If the matrix was found singular during factorization then one of the possible solutions is returned.

  i: with $1 \leq i \leq$ `INFOG(28)`. The i-th vector of the null space basis is computed.

 -1: The complete null space basis is computed.

Default value: 0 (normal solution step)

Incompatibility: Iterative refinement, error analysis, and the option to solve the transpose system (`ICNTL(9)` $\neq$ 1) are ignored when the solution step is used to return vectors from the null space (`ICNTL(25)` $\neq$ 0).

Related parameters: `ICNTL(21)`, `ICNTL(24)`

Remarks: Null space basis computation can be used when a zero-pivot detection option was requested (`ICNTL(24)` $\neq$ 0) during the factorization and when the matrix was found to be deficient (`INFOG(28)` > 0).

Note that when vectors from the null space are requested (`ICNTL(25)` $\neq$ 0), both centralized (`ICNTL(21)`=0) and distributed (`ICNTL(21)`=1) solutions options can be used. 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`, which must be allocated by the user on all working processors (see Subsection 5.13.5). In both cases, the number of columns of RHS or `SOL_loc` must be equal to the number of vectors requested, so that `NRHS` must be equal to:

- 1 if $1 \leq$ ICNTL(25) $\leq$ `INFOG(28)`
- `INFOG(28)` if ICNTL(25)=-1.

ICNTL(26) drives the solution phase if a Schur complement matrix has been computed (`ICNTL(19)` $\neq$ 0), see Subsection 3.16 for details

Phase: accessed by the host during the solution phase. It will be accessed also during factorization if the forward elimination is performed during factorization (`ICNTL(32)`=1)

Possible variables/arrays involved: REDRHS, LREDRHS

Possible values :

- 0 : standard solution phase on the internal problem; referring to the notations from Subsection 3.16, only the system $\mathbf{A}_{1,1}x_1 = b_1$ is solved and the entries of the right-hand side corresponding to the Schur are explicitly set to 0 on output.
- 1 : condense/reduce the right-hand side on the Schur. 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 the pointer array REDRHS, that must be allocated by the user. Its leading dimension LREDRHS must be provide, too.
- 2 : expand the Schur local solution on the complete solution variables. REDRHS is considered to be the solution corresponding to the Schur variables. It must be allocated by the user as well as its leading dimension LREDRHS must be provided. 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.

Default value: 0 (normal solution phase)

Incompatibility: if ICNTL(26) = 1 or 2, then error analysis and iterative refinement are disabled (`ICNTL(11)` and `ICNTL(10)`)

Related parameters: `ICNTL(19)`, `ICNTL(32)`

Remarks: If ICNTL(26) $\neq$ 0, then the user should provide workspace in the pointer array REDRHS, as well as a leading dimension LREDRHS (see Subsection 5.14). Note that if no Schur complement was computed, ICNTL(26) = 1 or 2 results in an error.

ICNTL(27) controls the blocking size for multiple right-hand sides.

Phase: accessed by the host during the solution phase

Possible variables/arrays involved: id%NRHS

Possible values :

< 0 : an automatic setting is performed by the solver:
   (i) the blocksize = min(id%NRHS, −2×ICNTL(27)) if the factors are on disk (`ICNTL(22)`=1);
   (ii) the blocksize = min(id%NRHS,−ICNTL(27)) if the factors are in-core (`ICNTL(22)`=0)

 0 : no blocking, it is treated as 1.

> 0 : blocksize = min(id%NRHS,ICNTL(27))

Default value: -32

Remarks: 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.

ICNTL(28) determines whether a sequential or parallel computation of the ordering is performed (see Subsection 3.2 and Subsection 5.4).

Phase: accessed by the host process during the analysis phase.

Possible values :

 0: automatic choice.
 1: sequential computation. In this case the ordering method is set by `ICNTL(7)` and the `ICNTL(29)` parameter is meaningless (choice of the parallel ordering tool).
 2: parallel computation. A parallel ordering and parallel symbolic factorization is requested by the user. For that, one of the parallel ordering tools (or all) must be available, and the matrix should not be too small. In this case `ICNTL(7)` is meaningless.

Any other values will be treated as 0.

Default value: 0 (automatic choice)

Incompatibility: The parallel analysis is not available when the Schur complement feature is requested (`ICNTL(19)`=1,2 or 3), when a maximum transversal is requested on the input matrix (i.e., `ICNTL(6)`=1, 2, 3, 4, 5 or 6) or when the input matrix is an unassembled matrices (`ICNTL(5)`=1). When the number of processes available for parallel analysis is equal to 1, or when the initial matrix is extremely small, a sequential analysis is indeed performed, even if `ICNTL(28)`=2 (no error is raised in that case).

Related parameters: `ICNTL(7)`, `ICNTL(29)`, `INFOG(32)`

Remarks: Performing the analysis in parallel (`ICNTL(29)` = 2) will enable saving both time and memory. Note that then the quality of the ordering depends on the number of processors used. The number of processors for parallel analysis may be smaller than the number of MPI processes available for MUMPS, in order to satisfy internal constraints of parallel ordering tools. On output, `INFOG(32)` is set to the type of analysis (sequential or parallel) that was effectively chosen internally.

ICNTL(29) defines the parallel ordering tool (when `ICNTL(28)`=1) to be used to compute the fill-in reducing permutation. See Subsection 3.2 and Subsection 5.4.

Phase: accessed by host process only during the parallel analysis phase (`ICNTL(28)`=2).

Possible variables/arrays involved: `SYM_PERM`

Possible values :

 0: automatic choice.
 1: PT-SCOTCH is used to reorder the input matrix, if available.
 2: ParMetis is used to reorder the input matrix, if available.

Default value: 0 (automatic choice)

Related parameters: ICNTL(28)

Remarks: On output, the pointer array SYM_PERM provides access, on the host processor, to the symmetric permutation that is effectively considered during the analysis phase, and INFOG(7) to the ordering option that was effectively used. SYM_PERM(i), (i=1, ... N) holds the position of variable i in the pivot order, see Subsection 5.4.1 for a full description.

ICNTL(30) computes a user-specified set of entries in the inverse $\mathbf{A}^{-1}$ of the original matrix (see Subsection 5.13.3).

Phase: accessed during the solution phase.

Possible variables/arrays involved: NZ_RHS, NRHS, RHS_SPARSE, IRHS_SPARSE, IRHS_PTR

Possible values :

  0: no entries in $\mathbf{A}^{-1}$ are computed.

  1: computes entries in $\mathbf{A}^{-1}$.

Other values are treated as 0.

Default value: 0 (no entries in $\mathbf{A}^{-1}$ are computed)

Incompatibility: Error analysis and iterative refinement will not be performed, even if the corresponding options are set (ICNTL(10) and ICNTL(11)). 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)). Option to compute solution using $\mathbf{A}$ or $\mathbf{A}^{T}$ (ICNTL(9)) is meaningless and thus ignored.

Related parameters: ICNTL(27)

Remarks: 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). Larger ICNTL(27) values will most likely decrease the amount of factor accesses, enable more parallelism and thus reduce the solution time [45, 41, 15].

The user must specify on input to a call of the solve phase in the arrays IRHS_PTR and IRHS_SPARSE the target entries. The array RHS_SPARSE should be allocated but not initialized. Note that since selected entries of the inverse of the matrix are requested, NRHS must be set to N. On output the arrays IRHS_PTR, IRHS_SPARSE and RHS_SPARSE will hold the requested entries. If duplicate target entries are provided then duplicate solutions will be returned.

When entries of $\mathbf{A}^{-1}$ are requested (ICNTL(30) = 1), mumps_par%**RHS** needs not be allocated.

ICNTL(31) indicates which factors may be discarded during the factorization.

Phase: accessed by the host during the analysis phase.

Possible values :

  0 : the factors are kept during the factorization phase except in the case of the out-of-core factorization of unsymmetric matrices when the forward elimination is performed during factorization (ICNTL(32) = 1). In this case, since it will not be used during the solve phase, the **L** factor is discarded: it is not written to disk.

  1: all factors are discarded during the factorization phase. 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 this case, the memory allocated for the factorization will rely on the out-of-core estimates (and factors will not be written to disk).

  2: this setting is meaningful only for unsymmetric matrices and has no impact on symmetric matrices: only the **U** factor is kept after factorization so that exclusively a backward substitution is possible during the solve phase (JOB=3). This can be useful when:

    −the user is only interested in the computation of a null space basis (see ICNTL(25))

during the solve phase, or

−the forward elimination is performed during the factorization (`ICNTL(32)`=1). Note that for unsymmetric matrices in out-of-core environments, if the forward elimination is performed during the factorization (`ICNTL(32)` = 1) then the **L** factor is always discarded during factorization. In this case (`ICNTL(32)` = 1), both `ICNTL(31)` = 0 and `ICNTL(31)` = 2 have the same behaviour.

Other values are treated as 0.

Default value: 0 (the factors are kept during the factorization phase in order to be able to solve the system).

Incompatibility: `ICNTL(31)` = 2 is not meaningful for symmetric matrices.

Related parameters: `ICNTL(32)`, forward elimination during factorization, `ICNTL(33)`, computation of the determinant, `ICNTL(25)` computation of a null space basis, `ICNTL(22)` out-of-core factors.

Remarks: For unsymmetric matrices, MUMPS currently discards **L** factors only in the out-of-core case (even when `ICNTL(32)` = 2). In a future version, discarding the **L** factor in the in-core case as well when `ICNTL(32)`=2 (or when `ICNTL(31)`=0 and `ICNTL(32)`=1) may lead to a memory reduction during the factorization.

ICNTL(32) performs the forward elimination of the right-hand sides (Equation (3)) during the factorization (`JOB`=2). (see Subsection 5.12).

Phase: accessed by the host during the analysis phase.

Possible variables/arrays involved: RHS, `NRHS`, `LRHS`, and possibly `REDRHS`, `LREDRHS` when `ICNTL(26)`=1

Possible values :

   0: standard factorization not involving right-hand sides.

   1: forward elimination (Equation (3)) of the right-hand side vectors is performed during factorization (`JOB`=2). The solve phase (`JOB`=3) will then only involve backward substitution (Equation (4)).

Other values are treated as 0.

Default value: 0 (standard factorization)

Related parameters: `ICNTL(31)`,`ICNTL(26)`

Incompatibility: This option is incompatible with sparse right-hand sides (`ICNTL(20)`=1,2,3), with the solution of the transposed system (`ICNTL(9)` $\neq$ 1), and with the computation of entries of the inverse (`ICNTL(30)`).

Furthermore, iterative refinement (`ICNTL(10)`) and error analysis (`ICNTL(11)`) are disabled. Finally, the current implementation imposes that all right-hand sides are processed in one pass during the backward step. Therefore, the blocking size (`ICNTL(27)`) is ignored.

Remarks: The right-hand sides must be dense to use this functionality: RHS, `NRHS`, and `LRHS` should be provided as described in Subsection 5.13.1. They should be provided at the beginning of the factorization phase (`JOB`=2) rather than at the beginning of the solve phase (`JOB`=3).

For unsymmetric matrices if the forward elimination is performed during factorization (`ICNTL(32)` = 1), the L factor (see `ICNTL(31)`) may be discarded to save space. In fact for unsymmetric matrices in out-of-core environments, if the forward elimination is performed during the factorization (`ICNTL(32)` = 1) then the L factors are always discarded during factorization even when `ICNTL(31)` = 0.

We advise to use this option only for a reasonable number of dense right-hand side vectors because of the additional associated storage required when this option is activated and the number of right-hand sides is large compared to `ICNTL(27)`.

ICNTL(33) computes the determinant of the input matrix.

Phase: accessed by the host during the factorization phase.

Possible values :

0 : the determinant of the input matrix is not computed.

$\neq 0$: computes the determinant of the input matrix. The determinant is obtained by computing $(a + ib) \times 2^c$ where $a =$`RINFOG(12)`, $b =$`RINFOG(13)` and $c =$ `INFOG(34)`. In real arithmetic $b=$`RINFOG(13)` is equal to 0.

Default value: 0 (determinant is not computed)

Related parameters: `ICNTL(31)`

Remarks: In case a Schur complement was requested (see `ICNTL(19)`), elements of the Schur complement are excluded from the computation of the determinant so that the determinant is that one of matrix $A_{1,1}$ (using notations of Subsection 3.16).

Although we recommend to compute the determinant on non-singular matrices, null pivots (`ICNTL(24)`) and static pivots (`CNTL(4)`) are excluded from the determinant so that a non-zero determinant is still returned on singular or near-singular matrices. This determinant is then not unique and will depend on which equations were excluded.

Furthermore, we recommend to switch off scaling (`ICNTL(8)`) in such cases. If not (`ICNTL(8)` $\neq 0$), we describe in the following the current behaviour of the package:

– if static pivoting (`CNTL(4)`) is activated: all entries of the scaling arrays `ROWSCA` and `COLSCA` are currently taken into account in the computation of the determinant.

– if the null pivot detection (`ICNTL(24)`) is activated, then entries of `ROWSCA` and `COLSCA` corresponding to pivots in `PIVNUL_LIST` are excluded from the determinant so that

* for symmetric matrices (`SYM`=1 or 2), the returned determinant correctly corresponds to the matrix excluding rows and columns of `PIVNUL_LIST`.

* for unsymmetric matrices (`SYM`=0), scaling may perturb the value of the determinant in case off-diagonal pivoting has occurred (`INFOG(12)` $\neq$0).

Note that if the user is interested in computing only the determinant, we recommend to discard the factors during factorization `ICNTL(31)`.

ICNTL(34) is not used in the current version.

ICNTL(35) controls the activation of the BLR feature (see Subsection 5.15).

Phase: accessed by the host during the analysis phase

Possible values :

0 : Standard multifrontal factorization (BLR is not activated)

1 : BLR approximation is activated (precision contoled by `CNTL(7)`).

Other values are treated as 0.

Default value: 0 (standard multifrontal factorization)

Related parameters: `CNTL(7)`

Incompatibility: For the number of operations performed when BLR is activated see `RINFOG(14)`. Note that the activation of BLR feature is currently incompatible with elemental matrices (`ICNTL(5)` = 1) and with forward elimination during factorization (`ICNTL(32)` = 1).

ICNTL(36-40) are not used in the current version.

6.2 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. See Subsection 3.9

Phase: accessed by the host during the factorization phase.

Possible values :

< 0.0: values less than 0.0 are treated as 0.0

$= 0.0$: no numerical pivoting performed and the subroutine will fail if a zero pivot is encountered.

> 0.0: numerical pivoting performed.
   For unsymmetric matrices values greater than 1.0 are treated as 1.0
   For symmetric matrices values greater than 0.5 are treated as 0.5

Default value:

 0.01: for unsymmetric or general symmetric matrices

 0.0: for symmetric positive definite matrices

Related parameters: `CNTL(4)`

Remarks: 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.

Note that for *diagonally dominant matrix*, setting CNTL(1) to zero will decrease the factorization time while still providing a stable decomposition.

CNTL(2) is the stopping criterion for iterative refinement

Phase: accessed by the host during the solve phase.

Possible values :≥ 0
Values < 0 are treated as $\sqrt{\epsilon}$, where ϵ holds the machine precision and depends on the arithmetic version.

Default value: $\sqrt{\epsilon}$

Related parameters: `ICNTL(10)`, `RINFOG(7)`, `RINFOG(8)`

Remarks: Let ω_1 and ω_2 be the backward errors as defined in Subsection 3.3.2. Iterative refinement (Subsection 5.5) will stop when either the requested accuracy is reached ($\omega_1 + \omega_2 < $ `CNTL(2)`) or when the convergence rate is too slow ($\omega_1 + \omega_2$ does not decrease by at least a factor of 5).

CNTL(3) it is used to determine if a pivot is null when the null pivot detection option is used (`ICNTL(24)` $= 1$)

Phase: accessed by the host during the numerical factorization phase.

Possible values :We define the threshold $thres$ as follows.

> 0.0: $thres = \text{CNTL(3)} \times \|\mathbf{A_{pre}}\|$

$= 0.0$: $thres = \epsilon \times 10^{-5} \times \|\mathbf{A_{pre}}\|$

< 0.0: $thres = |CNTL(3)|$

where A_{pre} is the preprocessed matrix to be factorized (see Equation (5)), ϵ is the machine precision and $\|.\|$ is the infinite norm.

Default value: 0.0

Related parameters: `ICNTL(24)`

Remarks: When null pivot detection is enabled (`ICNTL(24)`=1), `CNTL(3)` is the threshold used to determine if a pivot is null. A pivot is considered to be null if the infinite norm of its row/column is smaller than the threshold $thres$.

CNTL(4) determines the threshold for static pivoting. See Subsection 3.9

Phase: accessed by the host, and must be set either before the factorization phase, or before the analysis phase.

Possible values :

< 0.0: static pivoting is not activated.

= 0.0: static pivoting is activated and the threshold value to define a small pivot is determined automatically.

> 0.0: static pivoting is activated and the magnitude of small pivots smaller than CNTL(4) will be set to CNTL(4).

Default value: -1.0 (no static pivoting)

Related parameters: CNTL(1)

Remarks: By static pivoting (as in [37]) we mean replacing small pivots whose elimination should be postponed because of partial threshold pivoting and would thus result in an increase of our estimations (memory and operations), by a small perturbation of the original matrix controlled by CNTL(4).

CNTL(5) defines the fixation for null pivots and is effective only when null pivot detection is active (ICNTL(24) = 1).

Phase: accessed by the host during the numerical factorization phase.

Possible values :

≤ 0.0: In the symmetric case (SYM = 2) then the pivot column of the **L** factors is set to zero and the pivot entry in matrix **D** is set to one.

> 0.0: when a pivot piv is detected as null, in order to limit the impact of this pivot on the rest of the matrix, it is set to $sign(piv)$ CNTL(5) $\times \|\mathbf{A_{pre}}\|$, where $\mathbf{A_{pre}}$ is the preprocessed matrix to be factored (see Equation (5)). We recommend setting CNTL(5) to a large floating-point value (e.g. 10^{20}).

Default value: 0.0

Related parameters: ICNTL(24)

CNTL(7) defined the precision of the dropping parameter used during BLR (i.e. of the Low Rank approximation of the Blocks) (see Subsection 5.15).

Phase: accessed by the host during the factorization phase when ICNTL(35) = 1

Possible values :

0.0 : full precision approximation.

> 0.0 : the dropping parameter is **CNTL(7)**.

Default value: 0.0 (full precision (i.e., no approximation)).

Related parameters: ICNTL(35)

Remarks: Note that, depending on the application, gains can be expected even with small values (close to machine precision) of **CNTL(7)**.

CNTL(6 and 8-15) are not used in the current version.

6.3   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 2 highlights the incompatibilities between functionalities and matrix input formats (functionalities which do not appear in this table are compatible with all matrix input formats).

Functionality	(Control)	Matrix input format (ICNTL(18) and ICNTL(5))		
		Centralised		Distributed assembled
		Assembled	Elemental	(distr. elemental not avail.)
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
Block Low Rank	(ICNTL(35))	All options	Not available	All options

Table 2: 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
$\mathbf{LDL^t}$ factorization (SYM=2)	ok ok		ok but LU/PDGETRF performed on root node (no Scalapack $\mathbf{LDL^t}$ kernel)
Number of negative pivots (INFOG(12))	ok		lowerbound (negative pivots not counted on root node)

Table 3: 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.

In Table 3, 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 4. 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), or when the factors have been discarded during factorization.

Functionality	Control	iterative refinement `ICNTL(10)`	error analysis `ICNTL(11)`
Multiple right-hand sides	NRHS > 1	Incomp.	Incomp.
Distributed solution	`ICNTL(21)`	Incomp.	Incomp.
Forward during factorization	`ICNTL(32)`	Incomp.	Incomp.
Reduced right-hand sides/ partial solution	`ICNTL(26)`=1 `ICNTL(26)`=2	Incomp. Incomp.	Incomp. Incomp.
Discard some factors	`ICNTL(31)`	Incomp.	Incomp.
Compute null space (*)	`ICNTL(25)`	Incomp.	Incomp.
Entries of A^{-1}	`ICNTL(30)`	Incomp.	Incomp.

Table 4: List of incompatibilities with postprocessing options at the end of the solve phase. (*) The null space estimate is only available for symmetric matrices.

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 there is no incompatibility as such. But orderings based on minimum-degree (for example) are only available with the sequential analysis.

7 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 8), the information may be incomplete.

7.1 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 8), or positive if a warning is returned.

INFO(2) holds additional information about the error or the warning. If INFO(1) = $--1$, INFO(2) is the processor number (in communicator 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 (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 (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 $\boxed{in\text{-}core}$.

INFO(8) - after analysis: Minimum estimated size of the main internal real/complex workarray S to run the numerical factorization $\boxed{in\text{-}core}$. 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 (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 (millions of bytes) of all working space to run the numerical phases (factorization/solve) $\boxed{in\text{-}core}$ (ICNTL(22)=0 for the factorization). The maximum and sum over all processors are returned rerspectively 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. This excludes the memory for WK_USER, in the case where WK_USER is provided.

`INFO(17)` - after analysis: estimated size in MegaBytes (millions of bytes) of all working space to run the numerical phases $\boxed{\textit{out-of-core}}$ (`ICNTL(22)` $\neq 0$) 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 $\boxed{\textit{out-of-core}}$.

`INFO(20)` - after analysis: Estimated size of the main internal real/complex workarray S to run the numerical factorization $\boxed{\textit{out-of-core}}$. 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– or in the workarray `WK_USER`, in the case where `WK_USER` is provided. 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. This includes the memory effectively used in the workarray `WK_USER`, in the case where `WK_USER` is provided.

`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` $\times$ `NRHS` where `LSOL_loc` $\geq$ `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 (millions of bytes) 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) $\leq$ `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.

7.2 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(8)` - after solve with error analysis: Only returned if `ICNTL(11)` $= 1$ or 2. See description of `ICNTL(11)`.

`RINFOG(9)` to `RINFOG(11)` - after solve with error analysis: Only returned if `ICNTL(11)` $= 2$. 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)` - after factorization: The total number of floating-point operations (on all processors) for the elimination process when block low rank (BLR) functionality is activated.

RINFOG(15) - 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 last call to `MUMPS` was successful, negative if an error occurred (see Section 8), 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 null[12] 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 variable of the original matrix may be delayed several times between successive frontal matrices. In that case, it accounts for several delayed pivots. A large number (more that 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 $\boxed{\textit{in-core}}$.

- ——(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.

Note that in the case where WK_USER is provided, the memory allocated by the user for the local arrays WK_USER is not counted in INFOG(18) and INFOG(19).

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.

This includes the memory effectively used in the local workarrays WK_USER, in the case where the arrays WK_USER are provided.

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 $\boxed{\textit{out-of-core}}$ (ICNTL(22) $\neq$ 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.

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) $\leq$ INFOG(9).

---

[12]i.e., whose magnitude is smaller than the tolerance defined by CNTL(3).

`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.

8 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)`.

$--2$ `NNZ` (or `NZ`) is out of range. `INFO(2)`=NNZ (or `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 4. This error also occurs if `JOB` does not contain the same value on all processes on entry to `MUMPS`. `INFO(2)` is then set to the local value of `JOB`.

$--4$ Error in user-provided permutation array `PERM_IN` at position `INFO(2)`. This error may only occur on the host.

`−5` Problem of real workspace allocation of size `INFO(2)` during analysis. The unit for INFO(2) is the number of real values (single precision for SMUMPS/CMUMPS, double precision for DMUMPS/ZMUMPS), in the Fortran "ALLOCATE" statement that did not succeed. If `INFO(2)` is negative, then its absolute value should be multiplied by 1 million.

`−6` Matrix is singular in structure. `INFO(2)` holds the structural rank.

`−7` Problem of integer workspace allocation of size `INFO(2)` during analysis. The unit for INFO(2) is the number of integer values that MUMPS tried to allocate in the Fortran ALLOCATE statement that did not succeed. If `INFO(2)` is negative, then its absolute value should be multiplied by 1 million.

`−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` The 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)` or `LWK_USER` are provided, in which case `ICNTL(23)` or `LWK_USER` should be increased.

`−10` Numerically singular matrix.

`−11` Internal real/complex workarray S or `LWK_USER` too small for solution. If `INFO(2)` is positive, then the number of entries that are missing in S/LWK_USER at the moment when the error is raised is available in `INFO(2)`. If the numerical phases are out-of-core and `LWK_USER` is provided for the solution phase and is smaller than the value provided for the factorization, it should be increased by at least `INFO(2)`. In other cases, please contact us.

`−12` Internal real/complex workarray S too small for iterative refinement. Please contact us.

`−13` Problem of workspace allocation of size `INFO(2)` during the factorization or solve steps. The size that the package tried to allocate with a Fortran ALLOCATE statement 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. In general, the unit for `INFO(2)` is the number of scalar entries of the type of the input matrix (real, complex, single or double precision).

`−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. Normally, this error is raised on the sender side when detecting that the message to be sent is too large for the reception buffer on the receiver. `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
16	IRN_loc or JCN_loc

--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 INFOG(28); or ICNTL(25) was set to i, $1 \le i \le$ INFOG(28) and NRHS is different from 1. Value of NRHS is stored in INFO(2).

--33 ICNTL(26) was asked for during solve phase (or during the factorization – see ICNTL(32)) 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. This error also occurs in case the reduction phase (ICNTL(26) = 1) is asked for at the solution phase (JOB=3) but the forward elimination was already performed during the factorization phase (JOB=2 and ICNTL(32)=1). INFO(2) contains the value of ICNTL(26).

--36 Incompatible values of ICNTL(25) and INFOG(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.

--41 Incompatible value of `LWK_USER` between factorization and solution phases. This error may only occur when the factorization is in-core (`ICNTL(22)`=1), in which case both the contents of `WK_USER` and `LWK_USER` should be passed unchanged between the factorization (`JOB`=2) and solution (`JOB`=3) phases.

--42 `ICNTL(32)` was set to 1 (forward during factorization), but the value of `NRHS` on the host processor is incorrect: either the value of `NRHS` provided at analysis is negative or zero, or the value provided at factorization or solve is different from the value provided at analysis. `INFO(2)` holds the value of id%`NRHS` that was provided at analysis.

--43 Incompatible values of `ICNTL(32)` and ICNTL(xx). The index xx is stored in `INFO(2)`.

--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 `NRHS` $\leq$ 0. `INFO(2)` contains the value of `NRHS`.

--46 `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` $\geq$N, or `SIZE_SCHUR` was modified on the host since the analysis phase. The value of `SIZE_SCHUR` is provided in `INFO(2)`.

--50 An error occurred while computing the fill-reducing ordering during the analysis phase. This commonly happens when an (external) ordering tool returns an error code or a wrong result.

--51 An external ordering (Metis/ParMetis, SCOTCH/PT-SCOTCH, PORD), with 32-bit default integers, is invoked to processing a graph of size larger than $2^{31} - 1$. `INFO(2)` holds the size required to store the graph as a number of integer values; it is negative and its absolute value should be multiplied by 1 million.

--52 When default Fortran integers are 64 bit (e.g. Fortran compiler flag -i8 -fdefault-integer-8 or something equivalent depending on your compiler) then external ordering libraries (Metis/ParMetis, SCOTCH/PT-SCOTCH, PORD) should also have 64-bit default integers. `INFO(2)` = 1, 2, 3 means that respectively Metis/ParMetis, SCOTCH/PT-SCOTCH or PORD were invoked and were not generated with 64-bit default integers.

--53 Internal error that could be due to inconsistent input data between two consecutive calls.

--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)` $\geq$ 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 is close to zero. In some cases, this could cause difficulties in the computation of `RINFOG(6)`.

+4 Not used in current version.

+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.

9 Calling MUMPS from C

MUMPS is a Fortran 95 library, designed to be used from Fortran 95 rather than C. However a basic C interface is provided that allows users to call MUMPS directly from C programs. Similarly to the Fortran 95 interface, the C interface uses a structure whose components match those in the MUMPS structure for Fortran (Figure 2). Thus the description of the parameters in Sections 5 and 6 applies. Figure 4 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);
```

```
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;
     int64_t nnz;        /* Distributed entry */
     int nz_loc; int *irn_loc; int *jcn_loc; real/complex *a_loc;
     int64_t nnz_loc;        /* Element entry */
     int nelt; int *eltptr; int *eltvar; real/complex *a_elt;
     /* Ordering, if given by user */
     int *perm_in;
     /* Scaling */
     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 4: 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.

An example of calling MUMPS from C for a complex assembled problem is given in Subsection 11.3. 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.

9.1 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:NNZ)` in the Fortran version should be stored in `irn[0:nnz-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, ... NNZ} can be used instead of `id.irn[I-1]`; this notation then matches exactly with the description in Sections 5 and 6, where arrays are supposed to start at 1.

This can be slightly more confusing for elemental matrix input (see Subsection 5.2.2.3), 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]`.

9.2 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 = (MUMPS_INT) MPI_Comm_c2f(comm_c);
```

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

```
id.comm_fortran = (MUMPS_INT) comm_c;
```

should work.

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

9.3 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.

9.4 Runtime libraries

The Fortran 95 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`).

9.5 Integer, real and complex datatypes in C and Fortran

We assume that the `int`, `int64_t`, `float` and `double` types are compatible with the Fortran `INTEGER`, `INTEGER(KIND=8)`, `REAL` and `DOUBLE PRECISION` datatypes. Those assumptions are used in the files `[dscz]mumps_c_types.h`, which may need to be modified if needed.

Remark that Fortran compilers often provide an option to make all default `INTEGER` datatypes 64-bit integers. In that case, compiling C source files with the option `-DINTSIZE64` will consider that the default Fortran `INTEGER` matches a 64-bit integer of type `int64_t`.

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.

9.6 Sequential version

The C interface to `MUMPS` is compatible with the sequential version; see Subsection 3.11.

10 Scilab and MATLAB/Octave interfaces

Thanks to Octave MEX compatibility, an Octave interface can be generated based on the MATLAB one. The documentation provided in this section 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. 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`, `NNZ` (or `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`, `NZ_RHS`, ... 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`.

- In the out-of-core case, functionalities allowing to control the directory and name of temporary files, can only be controlled through the environment variables *MUMPS_OOC_TMPDIR* and *MUMPS_OOC_PREFIX* – see Subsection 5.7.

Please refer to the report [30] 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 95 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 do 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 recommand to swith printing off by setting id.ICNL(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. Also contains the nullspace in case of null space computation, or entries of the inverse, in case of computation of inverse entries.

Internal Parameters

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

11 Examples of use of MUMPS

11.1 An assembled problem

An example program illustrating a possible use of MUMPS on assembled DOUBLE PRECISION problems is given Figure 5.

Two files must be included in the program: mpif.h for MPI and mumps_struc.h for MUMPS. The file mumps_root.h must also be available because it is included in mumps_struc.h. The initialization and termination of MPI are performed in the user program via the calls to MPI_INIT and MPI_FINALIZE.

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

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

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

we could have as input

```
5               : N
12              : NNZ
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
```

```fortran
      PROGRAM MUMPS_EXAMPLE
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INCLUDE 'dmumps_struc.h'
      TYPE (DMUMPS_STRUC) mumps_par
      INTEGER IERR, I, I8
      CALL MPI_INIT(IERR)
C Define a communicator for the package.
      mumps_par%COMM = MPI_COMM_WORLD
C  Initialize an instance of the package
C  for L U factorization (sym = 0, with working host)
      mumps_par%JOB = -1
      mumps_par%SYM = 0
      mumps_par%PAR = 1
      CALL DMUMPS(mumps_par)
C  Define problem on the host (processor 0)
      IF ( mumps_par%MYID .eq. 0 ) THEN
        READ(5,*) mumps_par%N
        READ(5,*) mumps_par%NNZ
        ALLOCATE( mumps_par%IRN ( mumps_par%NNZ ) )
        ALLOCATE( mumps_par%JCN ( mumps_par%NNZ ) )
        ALLOCATE( mumps_par%A( mumps_par%NNZ ) )
        ALLOCATE( mumps_par%RHS ( mumps_par%N  ) )
        DO I8 = 1_8, mumps_par%NNZ
          READ(5,*) mumps_par%IRN(I8),mumps_par%JCN(I8),mumps_par%A(I8)
        END DO
        READ(5,*) ( mumps_par%RHS(I) ,I=1, mumps_par%N  )
      END IF
C  Call package for solution
      mumps_par%JOB = 6
      CALL DMUMPS(mumps_par)
      IF (mumps_par%INFOG(1).LT.0) THEN
       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
     &            "  mumps_par%INFOG(1)= ", mumps_par%INFOG(1),
     &            "  mumps_par%INFOG(2)= ", mumps_par%INFOG(2)
       GOTO 500
      END IF
C  Solution has been assembled on the host
      IF ( mumps_par%MYID .eq. 0 ) THEN
        WRITE( 6, * ) ' Solution is ',(mumps_par%RHS(I),I=1,mumps_par%N)
      END IF
C  Deallocate user data
      IF ( mumps_par%MYID .eq. 0 )THEN
        DEALLOCATE( mumps_par%IRN )
        DEALLOCATE( mumps_par%JCN )
        DEALLOCATE( mumps_par%A    )
        DEALLOCATE( mumps_par%RHS )
      END IF
C  Destroy the instance (deallocate internal data structures)
      mumps_par%JOB = -2
      CALL DMUMPS(mumps_par)
  500 CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 5: Example program using MUMPS on an assembled DOUBLE PRECISION problem

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

11.2   An elemental problem

An example of a driver to use `MUMPS` for element `DOUBLE PRECISION` problems is given in Figure 6.

The calling sequence is similar to that for the assembled problem in Subsection 11.1 but now the host reads the problem in components N, NELT, ELTPTR, ELTVAR, A_ELT, and RHS. Note that for elemental problems ICNTL(5) must be set to 1 and that elemental matrices always have a symmetric structure. For the two-element matrix and right-hand side

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

we could have as input

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

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

```
      PROGRAM MUMPS_EXAMPLE
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INCLUDE 'dmumps_struc.h'
      TYPE (DMUMPS_STRUC) mumps_par
      INTEGER I, IERR, LELTVAR, NA_ELT
      CALL MPI_INIT(IERR)
C Define a communicator for the package
      mumps_par%COMM = MPI_COMM_WORLD
C Ask for unsymmetric code
      mumps_par%SYM = 0
C Host working
      mumps_par%PAR = 1
C Initialize an instance of the package
      mumps_par%JOB = -1
      CALL DMUMPS(mumps_par)
C Define the problem on the host (processor 0)
      IF ( mumps_par%MYID .eq. 0 ) THEN
        READ(5,*) mumps_par%N
        READ(5,*) mumps_par%NELT
        READ(5,*) LELTVAR
        READ(5,*) NA_ELT
        ALLOCATE( mumps_par%ELTPTR ( mumps_par%NELT+1 ) )
        ALLOCATE( mumps_par%ELTVAR ( LELTVAR ) )
        ALLOCATE( mumps_par%A_ELT( NA_ELT ) )
        ALLOCATE( mumps_par%RHS ( mumps_par%N  ) )
        READ(5,*) ( mumps_par%ELTPTR(I) ,I=1, mumps_par%NELT+1 )
        READ(5,*) ( mumps_par%ELTVAR(I) ,I=1, LELTVAR )
        READ(5,*) ( mumps_par%A_ELT(I),I=1, NA_ELT )
        READ(5,*) ( mumps_par%RHS(I) ,I=1, mumps_par%N  )
      END IF
C Specify element entry
      mumps_par%ICNTL(5) = 1
C Call package for solution
      mumps_par%JOB = 6
      CALL DMUMPS(mumps_par)
      IF (mumps_par%INFOG(1).LT.0) THEN
       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
     &          "  mumps_par%INFOG(1)= ", mumps_par%INFOG(1),
     &          "  mumps_par%INFOG(2)= ", mumps_par%INFOG(2)
       GOTO 500
      END IF
C Solution has been assembled on the host
      IF ( mumps_par%MYID .eq. 0 ) THEN
        WRITE( 6, * ) ' Solution is ',(mumps_par%RHS(I),I=1,mumps_par%N)
C Deallocate user data
        DEALLOCATE( mumps_par%ELTPTR )
        DEALLOCATE( mumps_par%ELTVAR )
        DEALLOCATE( mumps_par%A_ELT )
        DEALLOCATE( mumps_par%RHS )
      END IF
C Destroy the instance (deallocate internal data structures)
      mumps_par%JOB = -2
      CALL DMUMPS(mumps_par)
 500  CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 6: Example program using MUMPS on an elemental DOUBLE PRECISION problem.

11.3 An example of calling MUMPS from C

An example of a driver to use MUMPS from C is given in Figure 7.

```c
/* 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;
  int64_t nnz = 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.nnz =nnz; 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]);
  }
  ierr = MPI_Finalize();
  return 0;
}
```

Figure 7: Example program using MUMPS from C on an assembled problem.

12   License

13   Credits

This version of MUMPS has been developed by employees of CERFACS, ENS Lyon,
INPT(ENSEEIHT)-IRIT, Inria and University of Bordeaux: Emmanuel Agullo,
Patrick Amestoy, Maurice Bremond, Alfredo Buttari, Philippe Combes, Marie Durand,
Aurelia Fevre, Abdou Guermouche, Guillaume Joslin, Jacko Koster, Jean-Yves L'Excellent,
Stephane Pralet, Chiara Puglisi, Francois-Henry Rouet, Wissam Sid-Lakhdar,
Tzvetomila Slavova, Bora Ucar and Clement Weisbecker.

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. 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.

[3] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015.

[4] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto. Fast 3D frequency-domain full waveform inversion with a parallel Block Low-Rank multifrontal direct solver: application to OBC data from the North Sea. *Geophysics*, 81(6):R363 – R383, 2016.

[5] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. On the complexity of the Block Low-Rank multifrontal factorization. 2016. submitted to SISC.

[6] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and scalability of the Block Low-Rank multifrontal factorization. In *Parallel Matrix Algorithms and Applications (PMAA'16)*, Bordeaux, France, July 2016.

[7] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Sparse direct solvers towards seismic imaging of large 3D domains. In *78th EAGE Conference, workshop methods and challenges of seismic wave modelling for seismic imaging*, Vienna, Austria, June 2016.

[8] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[9] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputer Applications*, 3:41–59, 1989.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):388–421, 2001.

[15] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. *SIAM Journal on Scientific Computing*, 34(4):A1975–A1999, 2012.

[16] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and F.-H. Rouet. Parallel computation of entries of A^{-1}. *SIAM Journal on Scientific Computing*, 37(2):C268–C284, 2015.

[17] P. R. Amestoy, I. S. Duff, D. Ruiz, and B. Uçar. A parallel matrix scaling algorithm. In J. M. L. M. Palma, P. R. Amestoy, M. J. Daydé, M. Mattoso, and J. C. Lopes, editors, *High Performance Computing for Computational Science, VECPAR'08*, number 5336 in Lecture Notes in Computer Science, pages 309–321. Springer-Verlag, 2008.

[18] 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.

[19] P. R. Amestoy, J.-Y. L'Excellent, F.-H. Rouet, and W. M. Sid-Lakhdar. Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver. In *High Performance Computing for Computational Science, VECPAR 2014 - 11th International Conference, Eugene, Oregon, USA, June 30 - July 3, 2014, Revised Selected Papers*, pages 156–169, 2014.

[20] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM Press, Philadelphia, PA, third edition, 1995.

[21] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10(2):165–190, 1989.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.

[33] 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.

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

[35] J.-Y. L'Excellent. *Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects*. Habilitation à diriger des recherches, École normale supérieure de Lyon, Sept. 2012. http://tel.archives-ouvertes.fr/tel-00737751.

[36] J.-Y. L'Excellent and M. W. Sid-Lakhdar. A study of shared-memory parallelism in a multifrontal solver. *Parallel Computing*, 40(3-4):34–46, 2014.

[37] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, 2003.

[38] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[39] F. Pellegrini. SCOTCH and LIBSCOTCH 5.0 User's guide. Technical Report, LaBRI, Université Bordeaux I, 2007.

[40] 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.

[41] F.-H. Rouet. *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides*. PhD thesis, Institut National Polytechnique de Toulouse, Oct. 2012.

[42] 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.

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

[44] W. M. Sid-Lakhdar. *Scaling multifrontal methods for the solution of large sparse linear systems on hybrid shared-distributed memory architectures*. Ph.D. dissertation, ENS Lyon, Dec. 2014.

[45] 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.

[46] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.

[47] C. Weisbecker. *Improving multifrontal solvers by means of algebraic block low-rank representations*. PhD thesis, Institut National Polytechnique de Toulouse, http://ethesis.inp-toulouse.fr/archive/00002471/, Oct. 2013.