MUltifrontal Massively Parallel Solver
(MUMPS 5.9.2)
Users' guide *

November 13, 2026

Abstract

This document describes the Fortran 95 and C user interfaces to MUMPS5.9.2, a software package to solve sparse systems of linear equations, with many features. For some classes of problems, the complexity of the factorization and the memory footprint of MUMPS can be reduced thanks to the *Block Low-Rank (BLR)* feature.

We describe in detail the data structures, parameters, calling sequences, and error diagnostics. Basic example programs using MUMPS are also provided.

Recent features include improved multithreading using tree parallelism, improved null space detection with rank-revealing feature and new statistics about the magnitude of pivots.

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

Contents

1 Introduction **5**

2 Notes for users of other versions of MUMPS **7**
2.1 ChangeLog . . . 7
2.2 Binary compatibility . . . 8
2.3 Upgrading between minor releases . . . 8
2.4 Upgrading from MUMPS 5.8.2 to MUMPS 5.9.0 . . . 8
2.5 Upgrading from MUMPS 5.7.3 to MUMPS 5.8.0 . . . 8
2.6 Upgrading from MUMPS 5.6.2 to MUMPS 5.7.0 . . . 9
2.7 Upgrading from MUMPS 5.5.1 to MUMPS 5.6.0 . . . 9
2.8 Upgrading from MUMPS 5.4.1 to MUMPS 5.5.0 . . . 10
2.9 Upgrading from MUMPS 5.3.5 to MUMPS 5.4.0 . . . 10
2.10 Upgrading from MUMPS 5.2.1 to MUMPS 5.3.5 . . . 10
2.11 Upgrading from MUMPS 5.1.2 to MUMPS 5.2.1 . . . 10
2.12 Upgrading from MUMPS 5.0.2 to MUMPS 5.1.2 . . . 11
2.12.1 Changes on installation issues . . . 11
2.12.2 Selective 64-bit integer feature . . . 11
2.13 Upgrading from MUMPS 4.10.0 to MUMPS 5.0.2 . . . 12
2.13.1 Interface with the Metis and ParMetis orderings . . . 12
2.13.2 Interface with the SCOTCH and PT-SCOTCH orderings . . . 12
2.13.3 ICNTL(10): iterative refinement . . . 12
2.13.4 ICNTL(11): error analysis . . . 13
2.13.5 ICNTL(20): sparse right-hand sides . . . 13
2.13.6 ICNTL(4): Control of the level of printing . . . 13

3 Main functionalities of MUMPS 5.9.2 **13**
3.1 Input matrix format . . . 14
3.2 Preprocessing . . . 14
3.3 Post-processing facilities . . . 16
3.3.1 Iterative refinement . . . 16
3.3.2 Error analysis and statistics . . . 16
3.4 Single precision factorization in double precision MUMPS version . . . 17
3.5 Null pivot row detection . . . 17
3.6 Rank-revealing factorization . . . 17
3.7 Computation of a solution of a deficient matrix and of a null space basis . . . 17
3.8 Solving the transposed system . . . 17
3.9 Forward elimination during factorization . . . 17
3.10 Arithmetic versions . . . 18
3.11 Numerical pivoting . . . 18
3.12 The working host processor . . . 19
3.13 MPI-free version . . . 19
3.14 Multithreaded parallelism (possibly combined with MPI parallelism) . . . 19
3.15 Using the BLAS extension GEMMT for symmetric matrix-matrix products . . . 20
3.16 Out-of-core facility . . . 20
3.17 Determinant . . . 20
3.18 Computing selected entries of A^{-1} . . . 20
3.19 Schur complement with reduced/condensed right-hand side . . . 21
3.20 Block Low-Rank (BLR) multifrontal factorization . . . 22
3.20.1 BLR factorization variants and their complexity . . . 22
3.20.2 Reducing the memory consumption using BLR . . . 23
3.20.3 Adaptive precision BLR feature . . . 23
3.21 Save / Restore feature . . . 24
3.22 Stop on user request and progress information . . . 24
3.23 Cheap re-analysis limited to recomputing statistics . . . 24

4 User interface and available routines **24**

5 Application Program Interface **27**
5.1 Principal actions of MUMPS (JOB) . . . 27
5.1.1 Main phases of MUMPS: Initialization, Analysis, Factorization, Solve, Termination . . . 27
5.1.2 Values of JOB for Save, Restore . . . 29
5.1.3 Values of JOB for special use . . . 29
5.2 Version number . . . 30
5.3 Control of parallelism (COMM, PAR) . . . 30
5.4 Input Matrix . . . 30
5.4.1 Matrix type (SYM) . . . 30
5.4.2 Matrix format . . . 31
5.4.2.1 Centralized assembled matrix (ICNTL(5)=0 and ICNTL(18)=0). . 32
5.4.2.2 Distributed assembled matrix (ICNTL(5)=0 and ICNTL(18)=1,2,3). 33
5.4.2.3 Elemental matrix (ICNTL(5)=1 and ICNTL(18)=0). . . . 34
5.4.3 Writing the input matrix to a file . . . 35
5.5 Preprocessing: permutation to zero-free diagonal and scaling . . . 36
5.5.1 Permutation to a zero-free diagonal (ICNTL(6)) . . . 38
5.5.2 Scaling (ICNTL(6) or ICNTL(8)) . . . 38
5.6 Preprocessing: symmetric permutations . . . 38
5.6.1 Symmetric permutation vector (ICNTL(7) and ICNTL(29)) . . . 40
5.6.2 Given ordering (ICNTL(7)=1 and ICNTL(28)=1) . . . 41
5.7 Preprocessing: exploit compression of the input matrix resulting from a block format (ICNTL(15)$\neq$ 0) . . . 41
5.8 Post-processing: iterative refinement . . . 42
5.9 Post-processing: error analysis . . . 44
5.10 Single precision factorization in double precision MUMPS instance . . . 45
5.11 Out-of-core (ICNTL(22)) . . . 45
5.12 Workspace parameters (ICNTL(14) and ICNTL(23)) and user workspace . . . 46
5.13 Null pivot row detection (ICNTL(24)) . . . 50
5.14 Rank-revealing factorization (ICNTL(56)) . . . 51
5.15 Discard matrix factors (ICNTL(31)) . . . 52
5.16 Computation of the determinant (ICNTL(33)) . . . 52
5.17 Forward elimination during factorization (ICNTL(32)) . . . 53
5.18 Right-hand side and solution vectors/matrices . . . 54
5.18.1 Dense right-hand side (ICNTL(20)=0) . . . 56
5.18.2 Sparse right-hand side (ICNTL(20)=1,2,3) . . . 56
5.18.3 Distributed right-hand side (ICNTL(20)=10,11) . . . 57
5.18.4 A particular case of sparse right-hand side: computing entries of $\mathbf{A}^{-1}$ (ICNTL(30)=1) . . . 59
5.18.5 Centralized solution (ICNTL(21)=0) . . . 60
5.18.6 Distributed solution (ICNTL(21)=1) . . . 60
5.19 Schur complement with reduced/condensed right-hand side (ICNTL(19) and ICNTL(26)) . . . 61
5.19.1 Centralized Schur complement stored by rows (ICNTL(19)=1) . . . 62
5.19.2 Distributed Schur complement (ICNTL(19)=2 or 3) . . . 62
5.19.3 Centralized Schur complement stored by columns (ICNTL(19)=2 or 3) . . . 63
5.19.4 Using partial factorization during solution phase (ICNTL(26)= 0, 1 or 2) . . . 64
5.20 Block Low-Rank (BLR) feature (ICNTL(35) and CNTL(7)) . . . 66
5.20.1 MUMPS installation and BLR functionality . . . 66
5.20.2 BLR API . . . 66
5.20.3 BLR output: statistics . . . 69
5.20.4 Adaptive precision BLR . . . 70
5.21 Save (JOB=7) / Restore (JOB=8) feature . . . 70

5.21.1 Location and names of the save files . . . 70
5.21.2 Deletion of the save files (JOB=-3) . . . 71
5.21.3 Important remarks for the restore feature (JOB=8) . . . 71
5.21.4 Combining the Save/Restore feature with out-of-core . . . 72
5.21.5 Combining the Save/Restore feature with WK_USER . . . 72
5.21.6 Error management . . . 73
5.22 Setting the number of OpenMP threads by MUMPS(ICNTL(16)) . . . 73
5.23 Converting MPI processes to OpenMP threads (ICNTL(17)) . . . 73
5.24 Compact workarray id%S at the end of factorization phase . . . 74
5.25 Stop on user request and progress information . . . 75
5.26 Improved multithreading using tree parallelism (ICNTL(48)), so called $\mathcal{L}_0$-threads feature 75
5.27 Offload activities on GPUs (ICNTL(51)) . . . 76

6 Control parameters **78**
6.1 Integer control parameters . . . 78
6.2 Real/complex control parameters . . . 103
6.3 Compatibility between options . . . 105

7 Information parameters **107**
7.1 Information local to each processor . . . 107
7.2 Information available on all processors . . . 111

8 Error and warning diagnostics **116**

9 Calling MUMPS from C **122**
9.1 Array indices . . . 124
9.2 Issues related to the C and Fortran communicators . . . 124
9.3 Fortran I/O . . . 124
9.4 Runtime libraries . . . 125
9.5 Integer, real and complex datatypes in C and Fortran . . . 125
9.6 Sequential version . . . 125

10 Scilab and MATLAB/Octave interfaces **125**

11 Accessing control parameters from a configuration file **127**

12 Examples of use of MUMPS **129**
12.1 An assembled problem . . . 129
12.2 An elemental problem . . . 129
12.3 An example of calling MUMPS from C . . . 133
12.4 An example of calling MUMPS from fortran using the Save/Restore feature and Out Of Core 134
12.5 An example of calling MUMPS from C using the Save/Restore feature . . . 137

13 License **141**

14 Credits **141**

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 [12, 13, 16, 31, 32, 39, 50, 37, 38, 21, 1, 55, 18, 51, 8, 44, 57, 22, 45, 54] 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** [47], 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 $\mathbf{x_{pre}}$ 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** substitution 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 processes. 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 3.9), 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 substitution 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 [56] for message passing and OpenMP for multithreading. It makes use of the BLAS [26, 27], LAPACK, BLACS, and ScaLAPACK [25] 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 3.12).

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 [53], SCOTCH [49] or Metis [41] (strongly recommended), and the possibility for the user to input a given ordering.
- scaling of the original matrix
- out-of-core capability
- save and restore feature
- detection of null pivot rows, basic estimate of rank deficiency and computation of a null space basis, improved null space detection with rank-revealing feature
- 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
- exploitation of a Block Low-Rank (BLR) format for factorization and solution possibly with adaptive precision representation of the low rank blocks
- single precision factorization and solve in double precision MUMPS
- offload activities on GPUs
- cheap re-analysis limited to recomputing statistics

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 website for recommendations, from our users, on how to use the solver on Windows platforms.

2 Notes for users of other versions of MUMPS

MUMPS 5.9.2 release incorporates new features and 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 5.9.2 and if you would like to upgrade your application to use MUMPS 5.9.2.

We discuss binary compatibility in Subsection 2.2 and how to upgrade between minor versions in Subsection 2.3. We then discuss in Subsection 2.12 and in Subsection 2.13 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 5.9.2.

The interface is backward compatible so that a code working with the previous version should also work with the new one (all codes including MUMPS headers should be recompiled). Still, the following points can be noted.

2.1 ChangeLog

```
Changes from 5.9.1 to 5.9.2
* Use sequential scaling code when omp_get_max_thread()=1
* Fixed segfault in case a call to PTSCOTCH ordering returns an error
* Fixed error RINFOG(14) (flops with BLR) always 0 even if printed value is correct
* Fixed MPI_IN_PLACE usage when -DAVOID_MPI_IN_PLACE is set
* Fixed erroneous pointer size during parallel analysis
* Fixed error during mixed precision deallocation of internal data
* Fixed uninitialised value during solution phase
* Limit use of OpenMP ATOMIC with multithreaded scaling (critical with ifx)
* Adjust memory estimations when taking into account early allocation of id%S array
* Fixed unsafe pointer manipulation in C inplace conversion/copy

Changes from 5.9.0 to 5.9.1
* GPU: fixed error introduced in 5.9.0 leading to less work exposed on GPU
* MPI_to_k_OMP: empty module added for Cmake build
* Print message when MUMPS is compiled with -DUSE_AVX512_VBMI
* Fixed syntax error detected during compilation with MSVC 14.44
* OpenMP: add missing !$ OpenMP sentinels to enable compilation without OpenMP
* Adaptive BLR precision: complete user guide with compiler options to be added
                          to OPTC to enable using AVX512-VBMI instruction set
* ICNTL(48)=1: fixed error when calling [SDCZ]MUMPS_DM_FREE_S_WK
               introduced since 5.8 releases
* Combining save/restore and WK_USER: clarify user guide
* Fixed: restore quantization of off-diagonal non-admissible BLR blocks
* Fixed: error -20 with single precision factorization and 64-bit integers
* Introduce compilation flag -Dnrm2_returns_double as a workaround  when
  snrm2 and scnrm2 BLAS functions return 64-bit instead of 32-bit reals (see INSTALL)


Changes from 5.8.2 to 5.9.0
* New feature: mixed/adaptive BLR precision feature (ICNTL(40))
* New feature: possibility to expose activities on GPUs  (ICNTL(51))
* New feature: Single precision factorization in double precision
  MUMPS instance (ICNTL(47))
* Use some MPI processes as OpenMP resources (ICNTL(17)
* Stop on user request and monitor progress information
* Cheap re-analysis to recompute statistics (JOB=13)
```

```
* Accessing control parameters from a configuration file
* Default value for BLR variant ICNTL(36) is 1 (0 in previous releases)
* Null-pivot: fixed nb_singular_values was not passed to c interface
*             fixed user guide, pivnul_list of size INFOG(28) and not N
*             INFOG(51) nb of null pivots detected with ICNTL(24)=1
* Fixed error introduced in 5.8.0 leading to incorrect value of INFOG(35)
* Scaling automatically activated with SYM=1 if ICNTL(35)>0 (BLR compression)
* Fixed overestimation of INFOG(21-22) with ICNTL(35)>0 (BLR compression)
* Workaround segfault in MUMPS_SETRVAL_ADDR_C on 32-bit address environments
```

2.2 Binary compatibility

In general, successive versions of the MUMPS package do not ensure binary compatibility. This means that you should recompile the source files of your application that include MUMPS include files when you switch to a new version of the MUMPS package. However, we normally ensure binary compatibility between minor releases (e.g., MUMPS x.y.z to MUMPS x.y.z').

2.3 Upgrading between minor releases

Between minor releases (e.g., MUMPS x.y.z to MUMPS x.y.z'), the number of source files (and general Makefile) may have changed. However, the interface is fully backward-compatible. Building and linking the new version of MUMPS with your application should thus be enough to use the latest version. Furthermore, if you use dynamic MUMPS libraries, you will not need to recompile your code.

2.4 Upgrading from MUMPS 5.8.2 to MUMPS 5.9.0

- The default value for the BLR factorization variant is 1 (ICNTL(36)=1) since it leads to a better tradoff between flops reduction and accuracy with respect to default value (ICNTL(36)=1) than in previous releases.
- With the default and automatic choice of scaling (ICNTL(8)=77) the input matrix will be scaled even with SYM= 1. This was only recommended to the user in previous releases.
- Adaptive BLR precision feature (ICNTL(40)) exploiting up to seven custom formats during double precision factorization.
- Single precision factorization in double precision MUMPS instances (ICNTL(47)=1, see Subsection 5.10).
- GPUs: possibility to expose activities on GPUs (ICNTL(51)) and exploit both NVIDIA and AMD accelerators with XKBlas $\geq$ 6, see Subsection 5.27.
- Cheap re-analysis to recomputing statistics (JOB= 13, see details in Subsection 3.23).
- Accessing control parameters from a configuration file. This feature gives access to MUMPS main control parameters (see Section 11).
- Use some MPI processes as OpenMP resources during the numerical phases (ICNTL(17), see further details in Subsection 5.23).

2.5 Upgrading from MUMPS 5.7.3 to MUMPS 5.8.0

- The BLR general behaviour has been slightly changed to increase compression
- In the examples, a dependency has been introduced to use single precision libraries when double precision libraries are used. This is because, in the future, single precision MUMPS routines will be called from double precision ones.
- Memory estimations in an MPI context is improved using a stricter proportional mapping algorithm.
- MPI communication buffer sizes are not anymore constrained by factored panel sizes.

- Enable different numbers of threads per MPI.
- Fixed error with binary format of WRITE_PROBLEM feature under WINDOWS.
- Fixed error with ICNTL(37)=1.

2.6 Upgrading from MUMPS 5.6.2 to MUMPS 5.7.0

- MUMPS now uses multithreading by default to exploit tree parallelism (ICNTL(48)=1): shared memory parallelism that was exploited only at a node parallelism level is now also exploited at the tree level. Although exploiting this improved multithreading improves performance in most cases, this feature can be switched off to obtain the same parallel approach as in previous versions by setting ICNTL(48)=0. It may be worth setting ICNTL(48)=0 in an out-of-core context and/or if memory usage is critical and is degraded by the additional parallelism from ICNTL(48)=1.
- Rank-revealing (ICNTL(56)): A rank-revealing feature postpones to the root node the pseudo-singularities and uses a singular value decomposition at the root node (see Subsection 3.6 and Subsection 5.14 for the description of the interface). This can be used to improve the numerical robustness of the null pivot detection (ICNTL(24)).
- The default value of CNTL(1) resulting from a call to MUMPS with JOB= –1 is now -1, meaning "automatic", instead of positive values equal to 0.01 or 0.0 (depending on the matrix type) in previous releases. This allows the threshold for numerical pivoting to be adapted dynamically when rank-revealing (ICNTL(56)) is later activated. When ICNTL(56) is 0 (rank-revealing not requested), the threshold resulting from the default value of CNTL(1) is identical to the one in the previous versions of the package. More details are available in the description of CNTL(1).
- In case of iterative refinement with the control parameter ICNTL(10) set to a posive value, the check for "too slow" convergence for iterative refinement (see Section 5.8 and Algorithm 2) was slightly modified, allowing iterative refinement to continue perform iterations when the convergence factor is at least 2 (instead of at least 5 in previous versions).
- Instead of crashing badly when the sizes of the Fortran and the C integers are incompatible (e.g. 32-bit Fortran and 64-bit in C or vice versa), an error is now raised, with new error code -69.
- In case of distributed right-hand side (IRHS_loc, RHS_loc with emtpy rows, the sparsity of the distributed RHS (Subsection 5.18) is now exploited internally. This typically leads to a reduction of the internal intermediate storage and of the number of operations during the forward substitution step.

2.7 Upgrading from MUMPS 5.5.1 to MUMPS 5.6.0

- The -DBLR_MT, indicating that the BLAS is compatible with OpenMP is deprecated. MUMPS now assumes by default that the BLAS library is compatible with OpenMP and that the number of threads in the BLAS adapts to the number of OpenMP threads available in the current OpenMP region. If this is not the case, then MUMPS should be compiled with OPTF containing -DBLR_NOOPENMP, indicating that the BLAS should not be called inside OpenMP parallel regions.
- Values of ICNTL(22) different from 1 (Out-Of-Core activated during factorization) are now treated as 0 (In-Core factorization), in previous versions they were treated as 1 (Out-Of-Core activated).
- The feature to discard factors (ICNTL(31)=1) now also discards the low-rank factors in case the block low-rank feature is activated (ICNTL(35)), leading to a smaller memory usage when ICNTL(31)=1 and ICNTL(35)=1 or 2. The feature to discard only the **L** factor (unsymmetric matrices, ICNTL(31)=2 or ICNTL(32)=1) now also discards the in-core **L** factor, whereas it was only effective for out-of-core factorizations in previous versions.
- Concerning the analysis by blocks, ICNTL(15) $<$ 0 is now authorized even if BLKPTR (or BLKVAR) is allocated (BLKPTR is then not accessed). As a consequence, error -57 with INFO(2)=4 no longer occurs.

2.8 Upgrading from MUMPS 5.4.1 to MUMPS 5.5.0

- Since the availability of the analysis by blocks feature, the symmetry ratio is no longer computed in case the analysis by blocks (ICNTL(15)) is activated. See the description of INFOG(8).
- The default value of ICNTL(38) has changed from 333 to 600. When not set by the user, this can thus make the memory estimates for the block low-rank factorization slightly more pessimistic.
- We advise using SCOTCH version 7.0.1 or later to benefit from a faster analysis phase (multithreading during SCOTCH).
- During the installation of MUMPS, the construction of the file mumps_int_def.h no longer needs a C compiler (this was required by some users for cross-compilation)

2.9 Upgrading from MUMPS 5.3.5 to MUMPS 5.4.0

- In case of null pivot detection (ICNTL(24)$\neq$ 0), CNTL(3) default value in previous releases ($\epsilon \times 10^{-5} \times \|\mathbf{A_{pre}}\|$) was too small and has been updated to $\epsilon \times \|\mathbf{A_{pre}}\| \times \sqrt{\mathbf{N_h}}$

2.10 Upgrading from MUMPS 5.2.1 to MUMPS 5.3.5

- RINFO(5), RINFO(6), RINFOG(15), RINFOG(16) were incorrect in the previous version and their values have been fixed.
- INFO(30) and INFO(31) on processes with MPI rank $>$ 0 are fixed.
- In case the block low-rank feature (ICNTL(35)$>$0), when numerical pivoting can be restricted (CNTL(1) = 0.0), then the use of ICNTL(36)=1 has improved, leading to more compression and better performance during factorization.
- Concerning the right-hand side format, it can now be provided distributed over the MPI processes, with either an arbitrary distribution or a distribution suggested by the package. This has been done thanks to an extension of the ICNTL(20) control parameter. We refer the reader to the description of ICNTL(20), and to Subsection 5.18.3 for more details on this feature.
- When using the C interface and 64-bit default integers, it is still necessary to compile MUMPS C files with -DINTSIZE64, but no longer necessary to use this flag to compile your application including MUMPS headers. The size of MUMPS_INT is defined in the MUMPS headers and it is now possible to test from an application how it was set during MUMPS installation (see the "INSTALL" file in the package and Subsection 9.5).

2.11 Upgrading from MUMPS 5.1.2 to MUMPS 5.2.1

Although MUMPS 5.2.1 contains new features compared to MUMPS 5.1.2 (one of them being the possibility to store factors in low-rank format, leading to significant memory gains on some classes of matrices), the installation procedure has not changed and the interface of existing features is almost fully backward compatible, subject to the remarks below. Still, your code should be recompiled.

A minor modification concerns the combination of the BLR feature (ICNTL(35)) with elemental input (ICNTL(5)) and with the forward elimination during factorization (ICNTL(32)). In 5.1.x versions of MUMPS, BLR was automatically switched off in case of elemental input or forward elimination during factorization where as an error is now raised in such cases (see error codes -43 and -800).

The scope of ICNTL(35) has been extended: ICNTL(35)=2 exploits low-rank factors during solve, ICNTL(35)=3 is a backward-compatibility option to only exploit low-rank factors during factorization but perform a full-rank solve, as was doing ICNTL(35)=1 in MUMPS 5.1.2. ICNTL(35)=1 now does an automatic choice of the "best" BLR option (currently BLR factorization and BLR solve).

Thanks to a more flexible and improved memory management, the ICNTL(23) parameter to limit the amount of memory allocation by MUMPS to a given amount is now compatible with the block low-rank feature (ICNTL(35)). Therefore, the error characterized by INFO(1)=-800 and INFO(2)=23 from previous versions has disappeared. Furthermore, when ICNTL(23) is provided, MUMPS will no

longer try to allocate all the memory authorized in virtual memory. This is due to the new capacity of dynamically allocating some working memory when the static workspace is not large enough.

In case a maximum amount of allowed memory ICNTL(23) is provided and MUMPS stops with an error because ICNTL(23) cannot be respected, a new specific error code -19 is now raised, whereas previously only the error code -9 was raised (in full-rank). This allows to distinguish between the two types of errors. The error code -9 may still occur from time to time to indicate that the main internal workarray allocated at the beginning of the factorization is too small (and as before, the parameter ICNTL(14) defining the relaxation of the main internal working space should be relaxed). However, this error should now occur much less often than before, e.g. in case of extreme numerical pivoting difficulties. This is because this version includes a first step towards a more flexible memory management that allows part of the working memory to be stored in dynamically allocated blocks when needed.

Because of dynamic allocations, remark that the parameter ICNTL(14) used to relax the main internal working space does not lead to a bound on the total memory allocated: dynamic data may in some cases be allocated when this allows to avoid -9 errors and low-rank matrices will also be allocated dynamically and possibly kept during the solve phase. If the user would like to provide a strict bound on the total allocated memory, we recommend the use of ICNTL(23).

Related to the low-rank feature, the new control parameter ICNTL(38) was introduced for the user to provide an estimate compression rate for the factors. It is used by the analysis phase to provide memory estimates for the block low-rank factorization.

Out-of-range values of ICNTL(12) are now treated as 1 (usual ordering) instead of 0 (automatic choice).

The range of statistics provided to the users was extended and the lengths of the arrays INFO and INFOG is now 80 instead of 40.

2.12 Upgrading from MUMPS 5.0.2 to MUMPS 5.1.2

2.12.1 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 BLR 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.

2.12.2 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 5.4.2).* 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 does not 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.

2.13 Upgrading from MUMPS 4.10.0 to MUMPS 5.0.2

2.13.1 Interface with the Metis and ParMetis orderings

Since the release of MUMPS 4.10.0, the Metis API has changed. MUMPS 5.9.2 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.

2.13.2 Interface with the SCOTCH and PT-SCOTCH orderings

MUMPS 4.10.0 was not compatible with SCOTCH 6.x. MUMPS 5.9.2 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.

2.13.3 ICNTL(10): iterative refinement

In both MUMPS 4.10.0 and MUMPS 5.9.2, 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 5.9.2, 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.

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

2.13.4 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 5.9.2, as shown in Table 1.

ICNTL(11) value	MUMPS 4.10.0 meaning	MUMPS 5.9.2 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 5.9.2 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.

2.13.5 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 5.9.2.

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 5.9.2 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 5.9.2 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.

2.13.6 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 5.9.2. 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).

3 Main functionalities of MUMPS 5.9.2

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 5.4.2.1 and Subsection 5.4.2.2, respectively for the assembled centralized and assembled distributed formats, and see Subsection 5.4.2.3 for the elemental format .

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

3.2 Preprocessing

During the analysis phase, the original matrix is preprocessed 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 :

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

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, [11]), an approximate minimum degree ordering with automatic quasi-dense row detection (QAMD, [7]), 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, [53]), the SCOTCH package [49] from the University of Bordeaux 1, and the Metis package from Univ. of Minnesota [41]. 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 5.6.2).

  – 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 [28, 29]. 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 [30], 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.

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

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 $\omega < \alpha$

It has been shown [24] 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 5.8).

The use of iterative refinement can be particularly useful if static pivoting has been used during factorization (see Subsection 3.11). It is also recommanded if a single precision factorization has been performed with a matrix in double precision, see Subsection 3.4.

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 [24]. 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 [24], 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 Single precision factorization in double precision MUMPS version

When using MUMPS with double precision arithmetic (i.e., DMUMPS or ZMUMPS), it is possible to perform a single precision factorization, without the user needing to cast the data (right-hand side, matrix, etc.) to single precision. In this case, storage and computational costs are decreased and the solution is returned in double precision. If requested, iterative refinement (see Subsection 3.3.1) is performed in double precision, leading to a double precision accuracy upon convergence [4]. This feature is controlled by ICNTL(47) (see Subsection 5.10). It can also be useful for iterative methods [3], where MUMPS (possibly using low rank approximations, static pivoting, ...) is used as a preconditioner.

3.5 Null pivot row detection

MUMPS gives the possibility to detect null pivot rows of a matrix during factorization. This option is controlled by ICNTL(24) (see Section Subsection 5.13). The number of null pivot rows provides an estimate of the rank deficiency. The computation of a solution of a deficient matrix and of a null space basis is then possible using the control parameter ICNTL(25) (see Subsection 3.7).

3.6 Rank-revealing factorization

MUMPS provides an experimental option for rank detection (controlled by ICNTL(56)) whose interface is fully described in Subsection 5.14. The computation of a solution of the deficient matrix and of a null space basis is then possible using the control parameter ICNTL(25) (see Subsection 3.7).

The problem of rank detection for the original sparse matrix is reduced to the problem of rank detection for the last dense frontal matrix associated to the root node of the elimination tree. In fact during factorization the numerical difficulties are postponed possibly up to the root node. At the root, a rank-revealing algorithm based a singular value decomposition is performed.

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

Using the ICNTL(25) parameter, MUMPS can compute one of the possible solutions of $AX = B$ if during factorization the matrix A has been found to be deficient if a zero-pivot detection option (Subsection 3.5) and/or rank-revealing (Subsection 3.6) have been requested.

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

3.8 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 3.9) is performed during the factorization, solving the transposed system is not allowed.

3.9 Forward elimination during factorization

It is possible to perform the forward elimination (Subsection 5.17) 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 5.15) 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 3.16), 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.

3.10 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,

3.11 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 [17].

3.12 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 5.3), 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.

3.13 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 5.3). 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 3.14).

3.14 Multithreaded parallelism (possibly combined with MPI parallelism)

MUMPS supports shared memory, multithreaded parallelism mostly through the use of multithreaded BLAS libraries such MKL, ACML or OpenBLAS. In case MPI is used, multithreading is used within each MPI process. 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, OPTC and OPTL variables of the MUMPS Makefile.inc file. This option is, for example, `-fopenmp` for GNU gfortran and `-qopenmp` for Intel ifort but equivalent options exist for other commonly used compilers.

The number of threads within the OpenMP parallel regions of MUMPS should be set through the OMP_NUM_THREADS environment variable. Note that, in most cases, the OMP_NUM_THREADS environment variable conveniently and efficiently controls both the number of threads in the BLAS library and in the MUMPS OpenMP parallel regions. Furthermore, it is worth to have an installation allowing BLAS calls inside OpenMP regions when using some features as the BLR feature (see Subsection 5.20) and the so called $\mathcal{L}_0$-threads feature (see Subsection 5.26). If the installation doesn't allow BLAS calls inside OpenMP regions please refer to the paragraph "Multithreading" in Subsection 5.20.1.

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.

3.15 Using the BLAS extension GEMMT for symmetric matrix-matrix products

MUMPS is able to use the GEMMT extension of the matrix-matrix product to update only the upper or the lower triangular part of the result matrix. To be compatible with MUMPS, the routine signature should be the same as the one described here: https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-fortran/top/blas-and-sparse-blas-routines/blas-like-extensions/gemmt.html.

We strongly recommend to use this ability if your BLAS library enables it. See the "Using BLAS extension GEMMT" section of the INSTALL to activate it.

3.16 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 3.9)).

3.17 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 5.15), significantly reducing the storage requirements.

3.18 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 [55, 51, 18]. If an LU factorization of A had been computed, a^{-1}_{ij}, 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 5.18.2). In a parallel environment it is not so natural to combine parallelism and exploiting sparsity. Work based on [51, 19] to exploit parallelism is provided in this release.

3.19 Schur complement with 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 5.19 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*:

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

  1. First solve

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

  2. Then, use the Schur matrix to solve

$$\begin{pmatrix} \mathbf{I} & 0 \\ 0 & \mathbf{S} \end{pmatrix} \begin{pmatrix} y_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{14}$$

  3. And thereafter solve

$$\begin{pmatrix} \mathbf{U}_{1,1} & \mathbf{U}_{1,2} \\ 0 & \mathbf{I} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ x_2 \end{pmatrix} \tag{15}$$

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. In the context of partial Schur factorization, y_2 vector is returned to the user and is often referred to as the reduced/condensed right-hand-side.
2. *Using the Schur matrix* :
   Thereafter 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.

3.20 Block Low-Rank (BLR) multifrontal factorization

A low-rank feature that allows decreasing the time and memory complexity of sparse direct solvers on some problems arising from partial differential equations is provided.

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 for example a truncated QR factorization. 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 [8, 9]) to reduce the memory footprint and the computational complexity of our multifrontal solver. The compression of BLR blocks is based on a truncated QR factorization with column pivoting.

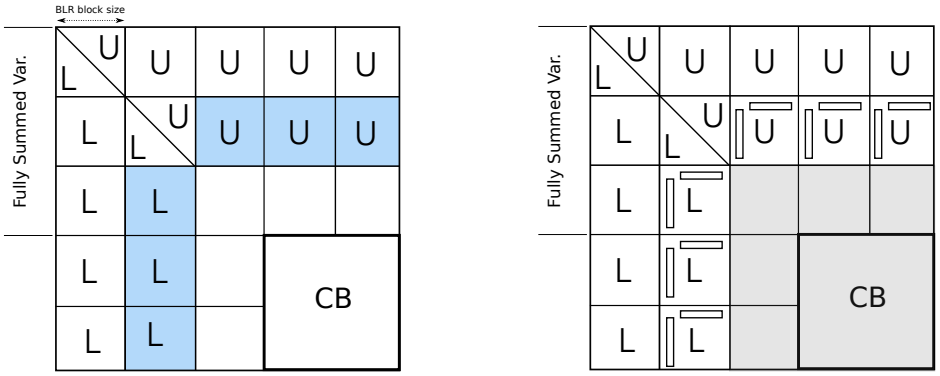
At each node of the multifrontal computational graph, a partial factorization (part corresponding to the fully summed variables, see Figure 1(a)) of a dense frontal matrix is performed. The so-called contribution block (CB in the figure) will correspond to the local Schur matrix built at the end of the 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 standard 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 BLR structure of the front. A truncated QR factorization, controlled by a numerical threshold (referred to as ε in Subsection 5.20), 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 trailing submatrix of the front (grey in (b)) at a lower cost than a full-rank standard update.

This functionality is controlled by a numerical parameter, called the low-rank threshold and noted ε, defining the accuracy of the low-rank approximations (see Subsection 5.20). This ε parameter can be used to control the backward error of the final solution [40]. With BLR compression, `MUMPS` can thus be used to provide either a direct solution at an accuracy controlled by ε, or an approximate factorization that can be used as a preconditioner (see [48]).

3.20.1 BLR factorization variants and their complexity

The theoretical analysis in [9] proves that the BLR approach can reduce the asymptotic complexity of the multifrontal factorization. The work presented in [9, 48] 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 by compressing at an earlier stage of the factorization.

The default choice of BLR variant in `MUMPS` is the left-looking standard UFSC factorization (described in [48]). It relies on the accumulation of the low-rank updates (so-called Low-rank Updates



(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.

Accumulation, or LUA) to improve the performance of the factorization by increasing the granularity and efficiency (Gflops/s) of the operations, as analyzed in [10].

The UFCS variant, described in [48], consists in performing the compression earlier to achieve a higher reduction of the number of operations. It is not used by default in the current version of MUMPS but can be activated (see Subsection 5.20). While its increased compression rate may slightly degrade the backward error, the currently implemented variant remains able to handle numerical pivoting and therefore insures numerical stability.

3.20.2 Reducing the memory consumption using BLR

The LU factors are compressed during the factorization. To reduce the memory footprint, factors are stored in low-rank form (see Subsection 5.20) . The solution phase is then performed using the low-rank factors and may thus also be accelerated. This is the automatic choice when BLR is activated.

It is also possible to keep the full-rank factors and discard the low-rank factors that have only thus be used to accelerate the factorization. The standard, full-rank solution phase is performed and no memory gains can be expected in this case.

The memory consumption can be further reduced by also compressing intermediate working space, the so-called contribution blocks (CB) of the frontal matrices, as described in [48]. However, contrarily to the LU factors compression, the CB compression does not contribute to reducing the global number of operations and therefore represents an overhead cost. For this reason, in the current version of MUMPS when BLR is activated the CB compression is not activated by default but can be activated (see Subsection 5.20). Note that in a parallel context, CB compression can also reduce the volume of communications.

3.20.3 Adaptive precision BLR feature

This work initiated in the context of the PhD thesis of Matthieu Gerest (LIP6-Sorbonne University and EDF R&D) [2, 35] is continued in the context of a post-doctorate of Antoine Jego [6, 5]. The memory footprint can be further reduced using adaptive precision BLR representation (see Subsection 5.20.4). This approach, proposed in [2], consists in partitioning the low-rank blocks into several groups of gradually decreasing precision, based on the decay of their singular values. It allows for maintaining a controlled backward error of order ε while reducing the size of the BLR LU factors via the use of lower precisions. A notable feature of this adaptive precision representation is that it can exploit an arbitrary number of precision formats. In the current version of MUMPS, we use seven precision formats – including custom formats with truncated mantissa – that maximize the storage reduction. Additionnally, blocks in BLR fronts that are not admissibile to BLR compression may be stored in their full-rank format in low precision if their norm is low (so called quantization of the off-diagonal blocks).

In order to maintain efficiency with these custom formats, we have developed a memory accessor approach [6, 5] that implements an efficient lightweight conversion from custom to standard formats. The efficiency of the approach relies on cache-aware mechanisms. When a computation is executed using (non-standard) low-precision operands, the operation is performed if said operand and a working precision copy of it can coexist in the L2 cache. If this is not possible, the operation is split up in smaller blocks: for instance a matrix-matrix multiply is split along the shared dimension of the operands.

For memory-bound computations involved in the solve phase, the reduced size of the BLR LU factors achieved by the use of lower precisions can lead to faster execution times.

3.21 Save / Restore feature

This version of MUMPS allows users to save MUMPS internal data before or after any of the main steps of MUMPS (analysis, factorization, solve, see Subsection 5.1.1) and then restart, possibly from another process (or set of MPI processes, in case of MPI), by restoring MUMPS internal data from disk and be at the same state as at the moment the data was saved to disk. After saving the data to disk, it is possible to continue working with the existing instance or terminate it. When the data are not needed anymore, MUMPS can be called to delete the data saved to disk. Examples of use are as follows:

- save all MUMPS internal data after factorization, terminate the instance, stop the process(es); later, start new process(es), initialize a new instance, and restore internal data from disk in order to continue from the point where data was saved to disk.
- similarly, save all MUMPS internal data after analysis; terminate the instance, stop (or not) the MPI processes; later, initialize a new instance, restore internal data and continue with the factorization phase.

More details on the Save and Restore feature (JOB= 7 and 8) and on the files deletion (JOB= –3) are provided in Subsection 5.1.1 and Subsection 5.21.

3.22 Stop on user request and progress information

MUMPS provides a feature to cancel computations and exit with an error, in case of the user finds the computations are too long and wants to cancel the factorization or the solve phase. Advancement can also be monitored during the factorization phase; it is provided as the number of floating-point operations done. The corresponding API details are provided in Section 5.25.

3.23 Cheap re-analysis limited to recomputing statistics

The use of $\mathcal{L}_0$-threads SMP parallelism may lead to an increase in the working memory which can be critical in case of Out-Of-Core (OOC) and BLR (BLR + OOC) factorization. This feature enables the user to change the values of ICNTL(47) (single precision factorization in double precision MUMPS instance), ICNTL(48) (improved multithreaded parallelism) and ICNTL(14) (percentage of memory relaxation), and to update all statistics performed during analysis while reusing all the main steps of a previous analysis phase (ordering, symbolic factorization, task pre-mapping and BLR clustering).

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]

[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.

The interface is the same for the MPI-free version (see Subsection 3.13), only the compilation process and libraries need be changed. In the case of the parallel version, MPI must be initialized by the user before the first call to [SDCZ]MUMPS is made.

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

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

```
      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(60)
         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, LRHS_loc, Nloc_RHS, LREDRHS, LSOL_loc, NSOL_loc
         real/complex, DIMENSION(:), POINTER :: SOL_loc, RHS_loc
         INTEGER, DIMENSION(:), POINTER :: ISOL_loc, IRHS_loc
C    Metis options, possibly modified by user
         INTEGER :: METIS_OPTIONS(40)
C OUTPUT data and Statistics
C --------------------------
         INTEGER, DIMENSION(:), POINTER :: SYM_PERM, UNS_PERM
         INTEGER INFO(80), INFOG(80) ! Local/global information
         real RINFO(40), RINFOG(40) ! Local/global information
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    Null pivots and rank-revealing
         INTEGER, DIMENSION(:), POINTER :: PIVNUL_LIST
         REAL, DIMENSION(:), POINTER :: SINGULAR_VALUES
         INTEGER ::  NB_SINGULAR_VALUES

C    Version number
         CHARACTER(LEN=30) VERSION_NUMBER
C    Name of file to dump a matrix/rhs to disk
         CHARACTER(LEN=1023) WRITE_PROBLEM
C    Out-of-core
         CHARACTER(LEN=255)  :: OOC_PREFIX
         CHARACTER(LEN=1023) :: OOC_TMPDIR
C    Save-Restore
        CHARACTER(LEN=1023) :: SAVE_DIR
        CHARACTER(LEN=255)  :: SAVE_PREFIX
      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 Principal actions of MUMPS (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:

- JOB= –1 initializes an instance of the package.
- JOB= –2 terminates an instance of the package.
- JOB= –3 Save / Restore feature: removes data saved to disk.
- JOB= –4 after factorization or solve phases, frees all MUMPS internal data structures except the ones from analysis.
- JOB= –200 (*experimental, subject to change in the future*) suppresses MUMPS out-of-core factor files associated to the calling MPI process and returns.
- JOB= 1 performs the analysis phase.
- JOB= 2 performs the factorization phase.
- JOB= 3 computes the solution.
- JOB= 4 combines the actions of JOB= 1 with those of JOB= 2.
- JOB= 5 combines the actions of JOB=2 and JOB= 3.
- JOB= 6 combines the actions of calls with JOB= 1, JOB= 2, and JOB= 3.
- JOB= 7 Save / Restore feature: saves MUMPS internal data to disk.
- JOB= 8 Save / Restore feature: restores MUMPS internal data from disk.
- JOB= 9 computes before the solution phase a possible distribution for the right-hand sides.
- JOB= 13 recomputes statistics provided during analysis while reusing all the main steps of a previous successful analysis phase. Incompatible with elemental matrix entry format ICNTL(5)=1.

5.1.1 Main phases of MUMPS: Initialization, Analysis, Factorization, Solve, Termination

JOB= **–1** initializes an instance of the package.

A call with JOB= –1 must be performed before any other call to the package on the same instance. It sets default values for other components of [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 3.12).

Finally, the version number is returned in VERSION_NUMBER (see Subsection 5.2).

In order to avoid memory leaks, an error −3 is raised if the user calls MUMPS with JOB= –1 twice on the same argument mumps_par without calling MUMPS with JOB= –2 in-between.

JOB= **–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. 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 except if the Save/Restore feature (Subsection 5.21) has been used and the files are associated to a saved instance. In this case, the out-of-cores files are kept because they will be exploited to restore the saved instance. See also Subsection 5.21.

JOB= **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 $\mathbf{A}$ input by the user. Several formats and distributions onto the processors are available to input matrix (see Subsection 5.4.2). 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 5.6) 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 5.5 and ICNTL(6)), then the numerical values of the original matrix $\mathbf{A}$ 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.

JOB= **2** performs the factorization.

It uses the numerical values of the matrix $\mathbf{A}$ provided by the user (see Subsection 5.4.2) and the information from the analysis phase (JOB= 1) to factorize the matrix $\mathbf{A}$.

The actual pivot sequence used during the factorization may slightly differ from the sequence returned by the analysis (see Subsection 5.6.1) if the matrix $\mathbf{A}$ is not diagonally dominant.

An option exists for the user to input scaling vectors or let MUMPS compute automatically such vectors (see Subsection 5.5.2 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.

JOB= **3** computes the solution.

It uses the right-hand side(s) $\mathbf{B}$ 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 $\mathbf{B}$ and to output the solution matrix $\mathbf{X}$ (see Subsection 5.18). The structure component mumps_par%RHS must be set by the user (on the host only) before a call with JOB= 3 (see Subsection 5.18).

This solution phase can also be used to compute the null space basis of singular matrices (see ICNTL(25)), provided that the rank-revealing option (ICNTL(56)) or null pivot row detection (ICNTL(24)) were on during factorization 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.

JOB= **4** combines the actions of JOB= 1 with those of JOB= 2.

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

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

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

5.1.2 Values of JOB for Save, Restore

JOB= **–3** Save / Restore feature: removes data saved to disk.
The files that were used to save an instance of MUMPS are deleted. It should be called by the user when no further restore of MUMPS with the files is required. See also Subsection 5.21.4.

JOB= **7** saves MUMPS internal data to disk.
It must be preceded by a call to MUMPS with JOB= –1 on the same instance. A call to MUMPS with JOB= 7 should be followed, at some point, by a call to MUMPS with JOB= –2 in order to free MUMPS internal data. The calling processes may then be stopped. It is possible to delete the files of a saved instance using a call to MUMPS with JOB= –3. See Subsection 5.21 for more details, in particular how to specify where to store the files of the saved instance.

JOB= **8** restores MUMPS internal data from disk.
It must be preceded by a call to MUMPS with JOB= –1 on the same instance. The values of PAR, SYM and COMM should be compatible with the values used at the moment the data have been written down to disk (JOB= 7). See Subsection 5.21 for more details, in particular where to find the files containing the saved instance.

5.1.3 Values of JOB for special use

MUMPS can be called with a special value of JOB between two phases of the computation. Such special values allow the user to get information to prepare the next steps.

JOB= **–200** (*experimental, subject to change in the future*) suppresses MUMPS out-of-core factor files associated to the calling MPI process and returns.
This feature was designed in case out-of-core was used (ICNTL(22)=1) and a serious crash occurred, otherwise it should not be used. For example, if MPI crashes with an error that cannot be recovered, or if some processes encounter some unrecoverable error that will lead to the termination of the processes, an error handler may call MUMPS with JOB= –200 in order to free the disk space associated with the out-of-core factor files before terminating the application. Internal data in memory are not freed and no MPI communications are performed during this call.

JOB= **–4** after factorization or solve phases frees all MUMPS internal data structures except the ones from analysis.
A call to MUMPS with JOB= –4 frees all MUMPS internal data structures except the ones from analysis. It can be used to free internal data allocated at factorization (factor matrices, ...) when the user wants to free some memory associated to an instance but still be able to perform a new factorization, possibly with new numerical values.

JOB= **9** before the solution phase computes a possible distribution for the right-hand sides.
Although the user is free to provide his/her own distribution of the right-hand sides, he/she could be interested in getting distributions targeting performance before calling the solve phase. Typically, a distribution that takes into account the internal distribution of the factor matrices will limit or suppress the data redistribution of the RHS on entry to the solution phase. A call to MUMPS with JOB= 9 can be done to obtain such distributions and can only be done after a successful factorization (JOB= 2 or 4 with INFOG(1) $\geq$ 0 on exit). On entry, IRHS_loc should be allocated on each MPI process, of size at least INFO(23) (i.e. the number of pivots eliminated on the processors). The result of the JOB= 9 call depends on the values on entry of ICNTL(20) = 10 or 11 and if the solution is going to be computed using $\mathbf{A}$ or $\mathbf{A}^T$ (ICNTL(9)). More information is provided in Subsection 5.18.3, which describes the distributed RHS interface.

JOB= **13** This feature enables to recompute the statistics performed during analysis. To be called analysis must have already been performed at least once. The user can change the values of ICNTL(47),

ICNTL(48) and ICNTL(14) and recompute all statistics while reusing all the main steps of the previous analysis phase (ordering, symbolic factorization, task pre-mapping and BLR clustering). See details in Subsection 3.23.

This feature is not compatible with elemental format (ICNTL(5)=1).

5.2 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 "5.9.2"`. 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).

5.3 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.

5.4 Input Matrix

5.4.1 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 $\mathbf{A}$ 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 $\mathbf{A}$ associated to the root node of the elimination tree. 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).

5.4.2 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 5.4.2.1) or in the structure components N, NNZ_loc (or NZ_loc), IRN_loc, JCN_loc, A_loc if the matrix is distributed (see Subsection 5.4.2.2).

1 : elemental format. The matrix must be input in the structure components N, NELT, ELTPTR, ELTVAR, and A_ELT (see Subsection 5.4.2.3).

Any other values will be treated as 0.

Default value: 0 (assembled format)

Related parameters: ICNTL(18)

Incompatibility: If the matrix is in elemental format (ICNTL(5)=1), the BLR feature (ICNTL(35) $\geq$ 1) is currently not available, see error -800.

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 5.4.2.1).

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 5.4.2.2).

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 5.4.2.2).

3 : user directly provides the distributed matrix, pattern and entries, input both for analysis and factorization (see Subsection 5.4.2.2).

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 recommend options 2 or 3. Among them, we recommend 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.

5.4.2.1 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 $\mathbf{A}$, 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.

5.4.2.2 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 5.4.2.1 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 $\mathbf{A}$, 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 $\mathbf{A}$, 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. In this case (ICNTL(18) = 1 or 2), they can be deallocated by the user just after the analysis phase.

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.

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.

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

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.

5.4.2.3 Elemental matrix (ICNTL(5) =1 and ICNTL(18) =0).

In the following, an example of elemental matrix with two elements is given.

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

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

- 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 $\mathbf{A}$, 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 $K_j + 1, \ldots, K_j + L_j$, where

→ $K_j = \sum_{p=1}^{j-1} {N_p}^2$, and $L_j = {N_j}^2$ in the unsymmetric case (SYM = 0)

→ $K_j = \sum_{p=1}^{j-1} (N_p \cdot (N_p + 1))/2$, and $L_j = (N_j \cdot (N_j + 1))/2$ in the symmetric case (SYM=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).

5.4.3 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 string "WRITE_PROBLEM" from the MUMPS structure. Either the "matrix market" format[4] or a binary format will be used to write the matrix, depending on the extension of "WRITE_PROBLEM" (see below). If a right-hand side is provided, it will also be written.

mumps_par%**WRITE_PROBLEM** (string, maximum length of 1023 characters) must be set by the user before the analysis phase (JOB= 1) in order to write the matrix/right-hand side into a file.

If the matrix is distributed, each MPI process must initialize the string WRITE_PROBLEM. Each MPI process then writes its share of the matrix in a file whose name is defined by the string "WRITE_PROBLEM" appended by its MPI rank.

Note that WRITE_PROBLEM should include both the path and the file name.

- If the last four characters of the string WRITE_PROBLEM are ".bin" (using either lowercase or uppercase letters), then a binary format is used. The write operations are performed using unformatted stream I/O, with no record boundaries. For a centralized matrix (ICNTL(18) =0), it contains:
  - N: four bytes (one 32-bit integer)
  - NNZ: eight bytes (one 64-bit integer)
  - IRN(1:NNZ): 4×NNZ bytes (NNZ 32-bit integers)
  - JCN(1:NNZ): 4×NNZ bytes (NNZ 32-bit integers)
  - A(1:NNZ) – if numerical values are provided at analysis: NNZ scalars, each of size 4 (resp. 8, 8, 16) bytes for s (resp. d, c, z) arithmetics.

  For a distributed matrix, the binary format is the same except that NNZ_loc, IRN_loc, JCN_loc, A_loc are printed on each MPI process (instead of NNZ, IRN, JCN, and A as above).
  A small text file which contains the corresponding matrix-market header and some comments describing the binary file is also written. The name of this text file is constructed by replacing ".bin" by ".header" in the string WRITE_PROBLEM.
- Otherwise, that is, if the last four characters of the string WRITE_PROBLEM different from ".bin", then the matrix (matrices in case of distributed format – ICNTL(18) =3) is written in matrix-market format.

Furthermore, if a dense and centralized right-hand side (see Subsection 5.18.1) is provided on the host before the analysis phase, it is also written in a file whose name is the matrix file name (WRITE_PROBLEM) appended by ".rhs". In case a binary format is used, the file for the RHS

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

contains N×NRHS scalars, of size 4, 8, 8, 16 for s, d, c, z arithmetics, respectively, and the header file mentioned above also provides the values of N and NRHS.

Finally, in case of analysis by blocks (see ICNTL(15)), BLKPTR and/or BLKVAR are also written if they are provided. If BLKPTR is provided, NBLK and BLKPTR(1:NBLK+1) are written (one integer per line, in text form) in a file whose name is the matrix file name appended by ".blkptr". If BLKVAR is provided, BLKVAR(1:N) is also written (one integer per line, in text form) in a file whose name is the matrix file name appended by ".blkvar".

5.5 Preprocessing: permutation to zero-free diagonal and scaling

The permutation to a zero-free diagonal and the scaling strategies 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 :

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 more costly (time and memory footprint) 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 if SYM= 0 or 2 (automatic choice done by the package),

0 if SYM= 1 (no column permutation is computed)

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 [28, 29]) 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 5.5.1). Otherwise, the pointer is not associated. The column permutation is such that entry $a_{i,uns_perm(i)}$ 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 [30] 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 done during analysis (see [28, 29] for the unsymmetric case and [30] for the symmetric case), under certain conditions. The original matrix has to be centralized (ICNTL(18)=0) and the user has to provide its numerical values (mumps_par%A) on entry to the analysis, otherwise the scaling will not be computed. Also, ICNTL(6) must be activated for the scaling to be computed.
The effective value of the scaling applied is reported in INFOG(33). We recommend that the user checks for INFOG(33) in order to know if the scaling was applied. When not applied, other scalings can be performed during the factorization.

-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 [52, 20, 43, 42]) 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

Other values are treated as 77.

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 real and symmetric with SYM= 1 then the automatic choice for scaling is no scaling except when BLR feature (ICNTL(35)) is activated since scaling is recommended previous to BLR factorization.

Incompatibility: 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 input matrix is assembled and distributed (ICNTL(18) =1,2,3 and ICNTL(5) = 0), then only options 7, 8 and 77 are allowed, otherwise no scaling is applied.
If block format is exploited (ICNTL(15) $\neq$ 0) then scaling is not applied.

5.5.1 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 [28, 29]) 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 is 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 $a_{i,uns_perm(i)}$ is on the diagonal of the permuted matrix.

5.5.2 Scaling (ICNTL(6) or ICNTL(8))

mumps_par%**ROWSCA**, mumps_par%**COLSCA** (**real** pointer arrays, dimension N) are optional, row and scaling scaling arrays, respectively, required only on the host. Note that these arrays are **real** also in the complex version. When allocated, ROWSCA(i) is the scaling factor of row i of the original matrix A, and COLSCA(j) is the scaling factor of column j of the original matrix A when no unsymmetric permutation occurred, and of matrix AQ_c (see Equation (5) in Subsection 3.2) when an unsymmetric permutation Q_c was performed on an unsymmetric matrix A (SYM=0, see ICNTL(6) and).

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.

5.6 Preprocessing: symmetric permutations

The choice of the symmetric permutation $\mathbf{P}$, 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. The ordering method is set by ICNTL(29) and the ICNTL(7) parameter is meaningless.

Any other values will be treated as 0.

Default value: 1 (sequential computation)

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(28) = 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) [11] 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] [49] package is used if previously installed by the user otherwise treated as 7.

4 : PORD[6] [53] is used if previously installed by the user otherwise treated as 7.

5 : the Metis[7] [41] package is used if previously installed by the user otherwise treated as 7.
It is possible to modify some components of the internal options array of Metis (see Metis manual) in order to fine-tune and modify various aspects of the internal algorithms used by Metis. This can be done by setting some elements (see the file metis.h in the Metis installation to check the position of each option in the array) of the MUMPS array mumps_par%**METIS_OPTIONS** after the MUMPS initialization phase (JOB= –1) and before the analysis phase. Note that the METIS_OPTIONS array of the MUMPS structure is of size 40, which is large enough for both Metis 4.x and Metis 5.x versions. It is passed by MUMPS as the argument "options" to the METIS ordering routine METIS_NodeND (METIS_NodeWND is sometimes also called in case MUMPS was installed with Metis 4.x) during the analysis phase.

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.

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

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 5.6.1.

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.

Other values are treated as 0.

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 5.6.1 for a full description.

5.6.1 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 5.19).

5.6.2 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 5.19). 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 5.6.1) 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.

5.7 Preprocessing: exploit compression of the input matrix resulting from a block format (ICNTL(15) $\neq$ 0)

The user can provide block information (so called block format) related to the input matrix, that will be exploited internally to work on a compressed graph during the ordering and symbolic factorization phases. This can significantly decrease the time and memory consumption for the analysis phase. Note that only the integer description of the block format is required.

If ICNTL(15) $\neq$ 0 then the block format will be described on the host process in the following components of [SDCZ]MUMPS_STRUC:

mumps_par%**NBLK** (integer) corresponds to the number of blocks and should be set by the user only if ICNTL(15) = 1

mumps_par%**BLKPTR** (integer pointer array, dimension NBLK+1) is such that BLKPTR(iblk) points to the position in BLKVAR of the first variable in block iblk. It must be set by the user only if ICNTL(15) = 1. If ICNTL(15) $<$ 0 it will be computed internally.

mumps_par%**BLKVAR** (integer pointer array, dimension N). If BLKVAR is not provided, it is internally treated as the identity BLKVAR(i)=i, (i=1, ..., N). Otherwise, BLKVAR(BLKPTR(iblk):BLKPTR(iblk+1)-1), (iblk=1, NBLK) holds the variables associated to block iblk.

ICNTL(15) exploits compression of the input matrix resulting from a block format

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

Possible variables/arrays involved: NBLK, BLKPTR, BLKVAR

Possible values :

0: no compression

-k: all blocks are of fixed size k>0. N (the order of the matrix A) must be a multiple of k. NBLK and BLKPTR should not be provided by the user and will be computed internally. Concerning BLKVAR, please refer to the *Remarks* below.

1: block format provided by the user. NBLK need be provided on the host by the user and holds the number of blocks. BLKPTR(1:NBLK+1) must be provided by the user on the host. Concerning BLKVAR, please refer to the *Remarks* below.

Any other values will be treated as 0.

Default value: 0

Remarks: If BLKVAR is not provided by the user then BLKVAR is internally treated as the identity (BLKVAR(i)=i, (i=1, ..., N)). It corresponds to contiguous variables in blocks.

- If ICNTL(15)=1 then BLKVAR(BLKPTR(iblk):BLKPTR(iblk+1)-1), (iblk=1, NBLK) holds the variables associated to block iblk.
- If ICNTL(15) <0 then BLKPTR need not be provided by the user and NBLK $= N/k$ where N must be a multiple of k.

In case the pivot order is provided on entry by the user at the analysis phase (ICNTL(7) $=1$) then PERM_IN should be compatible with the compression. This means that PERM_IN, of size N, should result from an expansion of a pivot order on the compressed matrix, i.e., variables in a block should be consecutive in the pivot order.

If the compression information does not introduce fill-in, the quality of the ordering should be similar with and without the activation of the analysis by blocks. If the quality of the ordering is degraded by the analysis by blocks, one should check that compression information is correct. On some types of matrices with many nonzeros per row where the analysis by blocks was observed to sometimes slightly degrade the quality of the ordering when using METIS (ICNTL(7)=5), note that setting the METIS option METIS_NSEPS to 2 (see) generally recovers a good ordering quality (while slightly increasing the time spent in METIS).

Incompatibility: With element entry format ICNTL(5)=1, with Schur complement ICNTL(19) $\neq 0$ and with permutation to a zero-free diagonal and related compressed/constrained ordering for symmetric matrices (ICNTL(6) $\neq 0$, ICNTL(12) $\neq 1$).

5.8 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. In the current implementation, "too slow" means that the backward error was divided by a factor less than 2 at the last iteration.

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 $r = b - Ax$
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 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 [24] 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.

Algorithm 3 Iterative refinement with a fixed number of steps 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

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

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), or if ICNTL(20)=10 or 11 (distributed right hand sides), then iterative refinement is disabled and 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 [24] 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 criterion (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 criterion (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 2)
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).

5.9 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 [24] (see Subsection 3.3.2).

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

- 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)
- ω_1 = RINFOG(7)
- ω_2 = RINFOG(8)

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)
- $\frac{\|\delta\bar{\mathbf{x}}\|_\infty}{\|\bar{\mathbf{x}}\|_\infty} \leq \omega_1\, cond_1 + \omega_2\, cond_2$ = 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 ICNTL(21)=1 (solution kept distributed) or if ICNTL(32)=1 (forward elimination during factorization), or if NRHS$>$1 (multiple right hand sides), or if ICNTL(20)=10 or 11 (distributed right hand sides), or if ICNTL(25)=-1 (computation of the null space basis), 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:

- 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).

5.10 Single precision factorization in double precision MUMPS instance

As indicated in Section Subsection 3.4, this feature allows the user to perform a single factorization in a double precision instance of MUMPS, and to perform iterative refinement in double precision. It is controlled by the parameter ICNTL(47), described below. The possibility to activate this feature has an impact on the compile and link phases of the package: when double precision (real or complex) version is requested, the single precision version is compiled first and the double precision version next, since the double precision version may depend on the single precision one.

For this feature to be enabled, the flag -DSINGLEINDOUBLE should be added to OPTF variable of the Makefile.inc used to compile MUMPS.

Furthermore, when -DSINGLEINDOUBLE has been added to OPTF, executables must be linked with both the double and the single precision versions of the MUMPS libraries.

ICNTL(47) controls the possibility to perform a single precision factorization in a double precision MUMPS instance

Phase: Accessed by the host during the analysis and factorization phases, only meaningful for double precision MUMPS instances (DMUMPS or ZMUMPS).

Possible values :

0 : nothing done, factorization and solves are performed in the working precision

1 : factorization and solves are performed in single precision

Out-of-range values are treated as 0.

Default value: 0 (factorization is performed in the working precision)

Incompatibility: This feature is not compatible with out-of-core execution (ICNTL(22)), with rank-revealing (ICNTL(56)), with the provision of a user workspace (WK_USER), and with the computation of a Schur complement (ICNTL(19)).

Remarks: The access to ICNTL(47) during the analysis phase only impacts the computed statistics. The value of ICNTL(47) is allowed to change between analysis and factorization, but memory statistics from the analysis will be incorrect. To obtain correct statistics with a new value of ICNTL(47), it is not necessary to perform the whole analysis again, performing a cheap analysis (JOB= 13) is enough.

5.11 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 3.16).

Other values are treated as 0 (in-core factorization).

Default value: 0 (in-core factorization)

Related parameters: ICNTL(35) since factors in low-rank form are not written to disk

Remarks: The variables OOC_TMPDIR and OOC_PREFIX are used to indicate the directory and the prefix, respectively, where to store the files containing 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 deallocation phase (JOB= –2 or JOB= –4) is called, except if the Save/Restore feature has been used and the files containing the factors are associated to a saved instance. See Section Subsection 5.21.4).

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.

5.12 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)). It also depends on the way factors of large frontal matrices are stored (full-rank or low-rank, see ICNTL(35) and ICNTL(37)).

Two main workarrays are allocated internally: IS and S (integer and real/complex workarray, respectively), that hold full-rank factors, active frontal matrices, and full-rank contribution blocks. Dynamic storage may also be used when the size of S was underestimated (ICNTL(14) too small). In case of low-rank storage of the factors (ICNTL(35)=2), dynamic allocation is used to store corresponding low-rank data. ICNTL(38) holds an estimate of the compression rate of the factors of BLR fronts on entry to the analysis phase and is be used to provide estimations of memory usage. In case of low-rank storage of CB blocks (ICNTL(37)=1) dynamic allocation is used to store corresponding low-rank data. ICNTL(39) holds an estimate of the compression rate of the CB blocks of BLR fronts on entry to the analysis phase and is used to provide estimations of memory usage.

In addition to these two large workarrays and to dynamic allocation for low-rank structures, other internal workarrays are used: for example, internal communication buffers in the parallel case, or integer arrays holding the structure of the elimination 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 depends on the memory relaxation parameter ICNTL(14) and in the case of low-rank storage it depends also of the average compression rate of the factors ICNTL(38) and of the average compression rate of the CB ICNTL(39). In case of out-of-core strategy, only uncompressed factors (in full-rank format) are written to disk and that MegaBytes corresponds to millions (10^6) of Bytes.

- **Full-rank factors** (ICNTL(35)=0 or 3):
  - Size in MegaBytes of the total working space locally requested by each processor:
    INFO(15) for in-core strategy;
    INFO(17) for out-of-core strategy.
    Maximum and sum over all processors:
    INFOG(16) and INFOG(17), respectively for in-core strategy;
    INFOG(26) and INFOG(27), respectively for out-of-core strategy;
  - Size of the main real/complex workarray S:
    INFO(8) for in-core strategy;
    INFO(20) for out-of-core strategy
    (negative value corresponds to *millions* of real/complex entries needed in this workarray).
- **Low-rank factors** (ICNTL(35)=1 or 2) only (ICNTL(37)=0) (estimates depends on both ICNTL(14) and ICNTL(38)):
  - Size in MegaBytes of the total working space locally requested by each processor:
    INFO(30) for in-core strategy;
    INFO(31) for out-of-core strategy.
    Maximum and sum over all processors:
    INFOG(36) and INFOG(37), respectively for in-core strategy;
    INFOG(38) and INFOG(39), respectively for out-of-core strategy;
  - Size of the main real/complex workarray S:
    INFO(29) for in-core strategy;
    INFO(20) for out-of-core strategy
    (negative value corresponds to *millions* of real/complex entries needed in this workarray).
- **Low-rank CB** only (ICNTL(35)=0,3 and ICNTL(37)=1) (estimates depends on both ICNTL(14) and ICNTL(39)):
  - Size in MegaBytes of the total working space locally requested by each processor:
    INFO(37) for in-core strategy;
    INFO(38) for out-of-core strategy.
    Maximum and sum over all processors:
    INFOG(44) and INFOG(45), respectively for in-core strategy;
    INFOG(46) and INFOG(47), respectively for out-of-core strategy;
  - Size of the main real/complex workarray S:
    INFO(36) for in-core strategy;
    INFO(33) for out-of-core strategy
    (negative value corresponds to *millions* of real/complex entries needed in this workarray).
- **Low-rank factors and CB** (ICNTL(35)=1,2 and ICNTL(37)=1) (estimates depends on ICNTL(14), ICNTL(38) and ICNTL(39)):
  - Size in MegaBytes of the total working space locally requested by each processor:
    INFO(34) for in-core strategy;

INFO(35) for out-of-core strategy.
Maximum and sum over all processors:
INFOG(40) and INFOG(41), respectively for in-core strategy;
INFOG(42) and INFOG(43), respectively for out-of-core strategy;

– Size of the main real/complex workarray S:
INFO(32) for in-core strategy;
INFO(33) for out-of-core strategy
(negative 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 by 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: between 20 and 35 (which corresponds to at most 35 % increase) and depends on the number of MPI processes. It is set to 5 % with SYM=1 and one MPI process.

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 process. It covers all internal integer and real (complex in the complex version) workspace allocated by MUMPS.

Phase: accessed by all processes at the beginning of the factorization phase. If the value is greater than 0 only on the host, then the value on the host is used for all processes, otherwise ICNTL(23) is interpreted locally on each MPI process.

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 process to be allocated

Default value: 0

Related parameters: ICNTL(14), ICNTL(38), ICNTL(39)

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 does not exceed 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 then shared between the main (and often most critical) real/complex internal workarray S holding the factors, the stack of contribution blocks and dynamic workarrays that are used either to expand the S array or to store low-rank dynamic structures.
Lower bounds for ICNTL(23), in case ICNTL(23) is provided only on the host:

– In case of full-rank factors only (ICNTL(35)=0 or 3), a lower bound for 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).

– In case of low-rank factors (ICNTL(35)=1 or 2) only (ICNTL(37)=0), a lower bound for ICNTL(23) (if ICNTL(14), has not been modified since the analysis and ICNTL(38) is a good approximation of the average compression rate of the factors) is given by INFOG(36) if the factorization is in-core (ICNTL(22)=0), and by INFOG(38) if the factorization is out-of-core (ICNTL(22)=1).

– In case of low-rank contribution blocks (CB) only (ICNTL(35)=0,3 and ICNTL(37)=1), a lower bound for ICNTL(23) (if ICNTL(14), has not been modified since the analysis and ICNTL(39) is a good approximation of the average compression rate of the CB) is given by INFOG(44) if the factorization is in-core (ICNTL(22)=0), and by INFOG(46) if the factorization is out-of-core (ICNTL(22)=1).
– In case of low-rank factors and contribution blocks (ICNTL(35)=1,2 and ICNTL(37)=1), a lower bound for ICNTL(23) (if ICNTL(14), has not been modified since the analysis, and ICNTL(38) and ICNTL(39) are good approximations of the average compression rate of respectively the factors and the CB) is given by INFOG(40) if the factorization is in-core (ICNTL(22)=0), and by INFOG(42) if the factorization is out-of-core (ICNTL(22)=1).

Lower bounds for ICNTL(23), in case ICNTL(23) is provided locally to each MPI process:

– Full-rank factors only (ICNTL(35)=0 or 3) $\Rightarrow$ INFO(15) if the factorization is in-core (ICNTL(22)=0), INFO(17) if the factorization is out-of-core (ICNTL(22)=1).
– Low-rank factors (ICNTL(35)=1 or 2) only (ICNTL(37)=0) $\Rightarrow$ INFO(30) if the factorization is in-core (ICNTL(22)=0), INFO(31) if the factorization is out-of-core (ICNTL(22)=1).
– Low-rank factors and contribution blocks (ICNTL(35)=1,2 and ICNTL(37)=1) $\Rightarrow$ is given by INFO(34) if the factorization is in-core (ICNTL(22)=0), INFO(35) if the factorization is out-of-core (ICNTL(22)=1).

The above lower bounds include memory for the real/complex internal workarray S holding the factors and stack of contribution blocks. In case WK_USER is provided, the above quantities should be diminished by the estimated memory for S/WK_USER. This estimated memory can be obtained from INFO(8), INFO(9), or INFO(20) (depending on MUMPS settings) by taking their absolute value, if negative, or by dividing them by 10^6, if positive. See also the paragraph *Recommended values of* LWK_USER below.

If ICNTL(23) is left to its default value 0 then MUMPS will allocate for the factorization phase a workspace based on the estimates computed during the analysis if ICNTL(14) has not been modified since analysis, or larger if ICNTL(14) was increased. Note that even with full-rank factorization, these estimates are only 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 larger than the provided estimations.

Another possibility, not recommended because of recent evolution of the solver as described below, is that the user provides the real/complex workarray instead of using the internal main real/complex workarray S. Note that in case factors of large frontal matrices are stored in low-rank form (ICNTL(35)=2) or in case of multithreading with tree parallelism, a separate dynamic storage allocation is performed. Providing a workarray to store frontal matrices, contribution blocks and factors that stay full-rank fronts and that are above the $\mathcal{L}_0$ layer (when ICNTL(48) $= 1$) is still possible even if less relevant in those cases because it might cover only a small part of the total workspace used. A pointer array that points to that workspace must be provided. In this case, the value of ICNTL(23) excludes the workspace corresponding to the user workspace.

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 define the size of the pointer array WK_USER. If negative, -LWK_USER is a lower bound for 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.

Recommended values of LWK_USER (otherwise an error with code −9 may occur):

- In case of full-rank factors (ICNTL(35)=0,3), we recommend the user to set LWK_USER to a value larger than than INFO(8) (in-core factorization) or INFO(20) (out-of-core factorization).
- In case of low-rank factors (ICNTL(35)=1 or 2) only (ICNTL(37)=0), we recommend the user to set LWK_USER to a value larger than INFO(29) (in-core factorization) or INFO(20) (out-of-core factorization).
- In case of low-rank contribution blocks only (ICNTL(35)=0,3 and ICNTL(37)=1), we recommend the user to set LWK_USER to a value larger than INFO(36) (in-core factorization) or INFO(33) (out-of-core factorization).

- In case of low-rank factors and contribution blocks (ICNTL(35)=1,2 and ICNTL(37)=1), we recommend the user to set LWK_USER to a value larger than INFO(32) (in-core factorization) or INFO(33) (out-of-core factorization).
- Finally, if improved multithreaded tree parallelism is activated (ICNTL(48) = 1, see Subsection 5.26), then the WK_USER/LWK_USER feature may be suboptimal and increase the memory peak. This is because in the current version of MUMPS, a large part of the memory will in that case be allocated in dynamically allocated thread-private data structures not exploiting WK_USER.

Moreover, if the factorization is in-core, 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.

5.13 Null pivot row detection (ICNTL(24))

It is possible to detect null pivot rows during the factorization by setting ICNTL(24) = 1. A pivot row is considered as null if its infinite norm is smaller than a threshold whose value relies on CNTL(3).

Null pivot row detection can be combined with rank-revealing feature (ICNTL(56) = 1). At the end of the factorization mumps_par%INFOG(28) will contain the deficiency of the initial matrix and if INFOG(28) $\neq$ 0, the array PIVNUL_LIST(1:INFOG(28)) will hold, on the host, the row indices corresponding to the null pivot rows and the singularities found at the root node (if the rank-revealing option has been requested (ICNTL(56) = 1)).

Null pivot rows are modified to enable the solution phase to provide one solution among the possible solutions of the numerically deficient matrix (see ICNTL(25) = 0) . CNTL(5) is used to define the value of the fixation. A vector or the complete basis of the null space can be returned to the user during the solve phase (see ICNTL(25) $\neq$0) using backward substitutions.

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 row will result in error INFO(1)=-10.

1: Null pivot row detection.

Other values are treated as 0.

Default value: 0 (no null pivot row detection)

Related parameters: CNTL(3), CNTL(5), ICNTL(13), ICNTL(25), ICNTL(56)

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

It can be used alone or combined with the rank-revealing option (see ICNTL(56)). In this case CNTL(3) it is used to determine if a pivot should be postponed.

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): INFOG(28) is set to the number of null pivot rows detected during the factorization step. Note that if the rank-revealing option (ICNTL(56)=1) is also activated INFOG(28) will be set to the total rank deficiency: null pivot rows plus deficiency of the root node of the elimination tree.

mumps_par%PIVNUL_LIST (integer array, dimension INFOG(28)):
If INFOG(28) $\neq$ 0 then PIVNUL_LIST(1:INFOG(28)) will hold, on the host, the row indices corresponding to the null pivot rows (ICNTL(24)= 1) and to the null singular values (ICNTL(56) = 1) found at the root node.

5.14 Rank-revealing factorization (ICNTL(56))

MUMPS has an experimental option for rank-revealing factorization.

During factorization, pseudo-singularities are postponed to the last dense frontal matrix (the so called *root node*). At the root node a rank-revealing factorization is performed based on the singular value decomposition. ScaLAPACK is automatically disabled (ICNTL(13)=-1). The elimination tree is preprocessed to reduce the size of root node.

During factorization pivots are considered as pseudo-singularities if their rows/columns have the infinite norm smaller than a threshold (relying on CNTL(3)).

To be able to determine the rank of the matrix, CNTL(3) is used to decide when a singular value (SVD decomposition on the root node if ICNTL(56) = 1) should be considered as null. The largest gap between singular values is then used to complete the list of null singular values which is returned in the array SINGULAR_VALUES of size NB_SINGULAR_VALUES.

Rank-revealing can be combined with null pivot row detection (ICNTL(24) = 1). A basis of the null space can be returned to the user by the solve phase (see ICNTL(25)) using backward substitutions.

ICNTL(56) detects pseudo-singularities during factorization, postpones them to the root node and factorizes the root node with a rank-revealing method.

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

Possible values :

0 : standard factorization is performed.

1 : Postponing and rank-revealing factorization on root node based on a singular value decomposition.

Values different from 1 are treated as 0.

Default value: 0 (standard factorization)

Related parameters: ICNTL(13), ICNTL(24), ICNTL(25), CNTL(1), CNTL(3)

Remarks: Note that ICNTL(56) must be set before analysis, during which a valid positive value prepares the data for later use of the rank-revealing functionality.

The root is processed sequentially and ICNTL(13) setting is ignored.

Please note that to improve the numerical behaviour of the factorization, the default value of CNTL(1) has been increased in case of activation of the rank-revealing feature. On numerically difficult problems the value of CNTL(1) may be further increased.

CNTL(3) it is used to determine if a pivot should be postponed.

Related data:

mumps_par%INFOG(28) (integer):
INFOG(28) is set during factorization to the deficiency of the matrix. It counts null pivot rows if ICNTL(24) = 1 and null singular values (ICNTL(56) = 1).

mumps_par%**PIVNUL_LIST** (integer array, dimension INFOG(28)):
If INFOG(28) $\neq$ 0 then PIVNUL_LIST(1:INFOG(28)) will hold, on the host, the row indices corresponding to both the null pivot rows (if ICNTL(24)= 1, the first entries upto INFOG(51) of PIVNUL_LIST) and the null singular values (ICNTL(56) = 1).

mumps_par%**SINGULAR_VALUES**(1:mumps_par%**NB_SINGULAR_VALUES**) (real pointer array) which holds, on the host, all the singular values corresponding to SVD decomposition on the root node.

If the matrix was found to be deficient (`INFOG(28)` > 0), the solution phase (`JOB= 3`) can then be used to either provide a "regular" solution or to compute the null-space basis (see `ICNTL(25)`).

Incompatibility: With Schur complement feature `ICNTL(19)` $\neq 0$

5.15 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 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.

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 a Schur complement is needed (see `ICNTL(19)`), or 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, 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 and `ICNTL(32)`=2, `MUMPS` currently discards the **L** factors corresponding to full-rank frontal matrices but not of low-rank frontal matrices (except if `ICNTL(35)`=3). In a future version, discarding all the **L** factors in case of BLR factorization and `ICNTL(32)`=2 may lead to a further memory reduction.

5.16 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 3.17 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 3.19).

Although we recommend to compute the determinant on non-singular matrices, null pivot rows (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 row 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).

5.17 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 perform the backward substitution, i.e., 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), with the computation of entries of the inverse (ICNTL(30)=1), and with BLR factorizations (ICNTL(35)=1,2,3). In such cases, error −43 is raised.

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 5.18.1. 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, the L factor will then always be discarded (even when ICNTL(31)=0) in the case of a full-rank factorization (ICNTL(35)=0) or BLR factorization with full-rank solve (ICNTL(35)=3). In the case of a BLR factorization with ICNTL(35)=1 or 2, only the L factor corresponding to full-rank frontal matrices are discarded in the current version.

We advise to use this option only for a reasonably small 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).

5.18 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 sparse format, centralized or distributed, 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 [18, 19, 51, 44]. 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 reduced when right-hand sides are sparse and provided in a sparse format.

Please note that, when not explicitly mentioned such as in paragraph "Special case of distributed right-hand side and distributed solution" of Subsection 5.18.3, input/ouptut arguments with different names should not point to the same memory location.

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 5.18.4).

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, sparse, or distributed) of the right-hand side.

Phase: accessed by the host during the solve phase and before a JOB= 9 call.

Possible variables/arrays involved: RHS, NRHS, LRHS, IRHS_SPARSE, RHS_SPARSE, IRHS_PTR, NZ_RHS, Nloc_RHS, LRHS_loc, IRHS_loc, RHS_loc, INFO(23).

Possible values :

0 : the right-hand side is in dense format in the structure component RHS, NRHS, LRHS (see Subsection 5.18.1)

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.

10, 11 : distributed right-hand side.
The right-hand side is provided distributed in the structure components Nloc_RHS, LRHS_loc, IRHS_loc, RHS_loc (see Subsection 5.18.3).
When provided before a JOB= 9 call, values 10 and 11 indicate which distribution MUMPS should build and return it to the user in IRHS_loc. In this case, the user should provide a workarray IRHS_loc on each MPI process of size at least INFO(23), where INFO(23) is returned after the factorization phase.

10 : fill IRHS_loc to minimize internal communications of right-hand side data during the solve phase.

11 : fill IRHS_loc to match the distribution of the solution in case of the distribution of the solution is imposed by MUMPS (ICNTL(21)=1).

In any case, values 10 and 11 have the same meaning entering the solve phase: use the distributed right-hand side in IRHS_loc, RHS_loc without taking into account how the distribution has been computed.

Values different from 0, 1, 2, 3, 10, 11 are treated as 0. For a sparse right-hand side, the recommended value is 1.

Default value: 0 (dense right-hand sides)

Incompatibility: When 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 5.18.1, Subsection 5.18.2 and Subsection 5.18.3. Please note that duplicate entries in the sparse or distributed right-hand sides are summed. A JOB= 9 call can only be done after a successful factorization phase and its results depends on the transpose option ICNTL(9), which should not be modified between a JOB= 9 and JOB= 3 call. The distributed right-hand side feature enables the user to provide a sparse structured RHS (i.e., a RHS with some empty rows that will be considered equal to zero).

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, SOL_loc, LSOL_loc, INFO(23).

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 5.18.5).

1 : the solution vector is kept distributed on each slave processor in the structure components ISOL_loc and SOL_loc. The distribution of the solution is chosen by MUMPS and is returned in ISOL_loc. The arrays ISOL_loc and SOL_loc must 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 5.18.6).

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.

5.18.1 Dense right-hand side (ICNTL(20)=0)

In this case, the matrix $\mathbf{B}$ 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 $\mathbf{A}$.

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 elimination and backward substitution 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). In case of forward elimination performed during factorization, NRHS should also be provided before analysis phase with JOB= 1 and should be kept unchanged for the subsequent numerical phases.

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, on the host processor, 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.

5.18.2 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 then be exploited to accelerate the solution phase (see [51, 44]) but at the cost of some extra preprocessing that can be switched off by the user setting ICNTL(20) to 2.

In the following, an example of a 4x2 matrix $\mathbf{B}$ with 5 nonzeros is provided.

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

total nonzeros in $\mathbf{B}$	NZ_RHS	= 5						
number of columns $\mathbf{B}$ (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) is an optional parameter that should be set by the user on the host processor, to the number of right-hand side vectors. Otherwise, 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).

5.18.3 Distributed right-hand side (ICNTL(20)=10,11)

In the context of dense multiple right-hand sides, the memory footprint on the host processor can be an issue. The associated communications can also be responsible for a non-negligible part of the cost of the solve phase. Furthermore, from a user's point of view it may be more natural in an MPI environment to provide the right-hand side already distributed on the MPI processors. The distributed right-hand feature has been developed to address these issues. It can also be used when the right-hand side is sparse but all columns have the same structure.

In order to activate the distributed right-hand side feature, the parameter ICNTL(20) should be set to 10 or 11. The user can be guided on how to distribute the right-hand side using the distribution returned by a call to MUMPS with JOB= 9. Such a distribution depends on the value of ICNTL(20) (10 or 11) chosen by the user (see paragraph "Special distributions returned by MUMPS" below).

Note that the solve phase of MUMPS will have exactly the same behaviour whether ICNTL(20)=10 or ICNTL(20)=11, and this whether a JOB= 9 call is performed or not, because MUMPS will use the distribution in IRHS_loc without taking into account any information on how the distribution has been computed.

Note that if the user wants to use the distribution returned by MUMPS after a JOB= 9 call (see paragraph "Special distributions returned by MUMPS" below), this can be done 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.

When the distributed right-hand side feature is requested (ICNTL(20)=10 or 11), the following parameters should be allocated and possibly set by the user before the solve phase (JOB= 3,5,6).

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

mumps_par%**Nloc_RHS** (integer) is the number of local right-hand side rows. It must be defined by the user 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). Note that Nloc_RHS=0 is allowed, indicating that the local processor does not contribute to the right-hand side and that Nloc_RHS=0 should not be set to a non-zero value on the non-working host (PAR=0).

mumps_par%**LRHS_loc** (integer) is the leading dimension of RHS_loc. It must be defined by the user 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) such that LRHS_loc $\geq$ Nloc_RHS.

mumps_par%**RHS_loc** (**real/complex** pointer array, dimension LRHS_loc×NRHS) must be allocated by the user on all processors if PAR=1, and on all processors except the host if PAR=0. The RHS_loc array should be filled by the user with the values of the right-hand side following the distribution provided in IRHS_loc (see below).

mumps_par%**IRHS_loc** (integer pointer array, of dimension Nloc_RHS) must be allocated and defined on all processors if PAR=1, and on all processors except the host if PAR=0. The user must set IRHS_loc(k) to the global index (between 1 and N) of the right-hand side rows that will be provided locally in RHS_loc. For example, if IRHS_loc(5)=47, row 5 of of the local RHS_loc array corresponds to row 47 in the global system. Note that if IRHS_loc(k_1)=IRHS_loc(k_2), for $k_1 \neq k_2$, or if the same index appears on several MPI processes, the corresponding rows in RHS_loc are summed. Furthermore, entries such that IRHS_loc(k) greater than N or smaller than 0 result in row k of RHS_loc to be ignored. Finally, it is not necessary that all global indices appear in some local IRHS_loc array. Providing only a few global indices means that the right-hand side is sparse, rows corresponding to global indices not provided will be considered to be equal to zero, and sparsity will be exploited to reduce the cost of the forward substitution step.

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). In this case, on exit from a call to MUMPS it will hold the centralized solution. Otherwise, it is not used and needs not be allocated.

Special distributions returned by MUMPS**.**

We now describe how to obtain special IRHS_loc distributions from MUMPS, that can guide the distribution of the right-hand sides provided by the user. This can be useful in case the user does not have specific constraints on the RHS distribution. For this, MUMPS should be called with JOB= 9 after a successful factorization, with the following parameters.

mumps_par%IRHS_loc (integer pointer array, of dimension at least INFO(23), where INFO(23) was computed during the factorization phase) must be allocated by the user after the factorization phase. On exit from a call to MUMPS with JOB= 9, IRHS_loc(1:INFO(23)) contains the list of global RHS indices on the local MPI process corresponding to the internal MUMPS distributions.

mumps_par%ICNTL(20)=10 if the user asks for a distribution that avoids RHS communication during MUMPS solve phase, or 11 if the user asks for a distribution that is identical to the distribution of the solution in case the distribution of the solution is imposed by MUMPS (ICNTL(21)=1). If, before a JOB= 9 call, ICNTL(20) is not set to 10 or 11, it is treated as 10.

mumps_par%ICNTL(9) indicates whether the distribution provided should target the standard solve phase ($\mathbf{AX} = \mathbf{B}$) or the solve phase for the transposed system ($\mathbf{A^T X} = \mathbf{B}$). Note that if ICNTL(9) changes between a JOB= 9 call and a JOB= 3 call, the distribution computed with ICNTL(20)=10 will be the one corresponding to the description of ICNTL(20)=11 and vice versa.

Remark 1: in some cases all possible values of ICNTL(9) and ICNTL(20) lead to the same distribution:

- in the symmetric case (SYM=1 or 2),
- in the unsymmetric case (SYM=0) when pivoting and maximum weighted matching options are switched off (CNTL(1)=0 and ICNTL(6)=0),
- in case an unsymmetric factorization (SYM=0) did not use an unsymmetric permutation of the matrix (see ICNTL(6)) and there were no off-diagonal pivots (INFOG(12)=0),

In such cases, the control parameters ICNTL(9) and ICNTL(20) need not be set on entry to the call to MUMPS with JOB= 9.

Remark 2: once a JOB= 9 call that defines IRHS_loc has been performed, the user is allowed to modify IRHS_loc to change the distribution suggested by MUMPS, e.g., to suppress rows that are known to be zero. Before calling the solve phase (JOB= 3), he/she should also provide the required arguments for JOB= 3 accordingly: Nloc_RHS, LRHS_loc and RHS_loc.

Special case of distributed right-hand side and distributed solution.

In case of both distributed RHS (ICNTL(20)=10,11) and distributed solution (ICNTL(21)=1), IRHS_loc and ISOL_loc may point to the same workarrays. This should only be done when the contents are known to be identical (symmetric case, unsymmetric case under some circumstances, see Remark 1 above). Otherwise, the solve phase (JOB= 3) with distributed solution will overwrite entries of IRHS_loc while building ISOL_loc.

Furthermore, RHS_loc and SOL_loc may point to the same memory location, as long as LRHS_loc > LSOL_loc.

5.18.4 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 3.18) 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 [55, 51, 18].

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.

5.18.5 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.

5.18.6 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, one possibility is to allow 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 if ICNTL(21)=1, JOB= 3,5,6 if ICNTL(21)=2).

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

mumps_par%**LSOL_loc** (integer) must be set to the leading dimension of SOL_loc.

It must be defined by the user on all MPI processes for which INFO(23), as returned by MUMPS during a JOB= 2 call (factorization phase), is nonzero and should be larger than or equal to INFO(23). Note that, in case of non-working host model of parallelism (PAR=0), LSOL_loc needs not be defined on the host on which INFO(23)=0.

mumps_par%**SOL_loc** (**real/complex** pointer array, dimension LSOL_loc× NRHS) must be allocated by the user before the solve step on MPI processes where LSOL_loc (see above) has been set to a value greater than 0. On exit from the solve phase, SOL_loc(iloc+(k-1)×LSOL_loc) will contain the value corresponding to variable ISOL_loc(iloc) in the k^{th} solution vector.

mumps_par%**ISOL_loc** (integer pointer array of dimension at least INFO(23)).

Since ISOL_loc should be of size at least INFO(23), with INFO(23) returned by the factorization phase, ISOL_loc should be allocated by the user between the factorization and the solve phases. 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 normally have the same contents for each of these calls. The only exception is the case of unsymmetric matrices (SYM=1) when the transpose option is changed (see ICNTL(9)) and non symmetric row/column exchanges (see ICNTL(6)) have occurred before the solve phase.

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.

5.19 Schur complement with reduced/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 5.19.1, Subsection 5.19.3 and Subsection 5.19.2).

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 5.19). 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: rank-revealing factorization (ICNTL(56)=1), maximum transversal, scaling, iterative refinement, error analysis and parallel analysis.

Related parameters: ICNTL(7), ICNTL(26)

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 5.19.3), 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 i^{th} 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 5.6.2, 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.

5.19.1 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 5.19.3 "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 $\times$ 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 $\times$ 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.

5.19.2 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 [25] (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$\times$ 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 5.19.3.

5.19.3 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 5.19.2), 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 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.

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

As explained in Subsection 3.19, 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 3.19 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 3.19, 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.

[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.

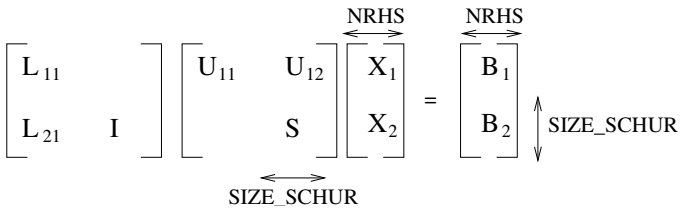


Figure 3: Solving the complete system using the Schur complement matrix

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 and 2 then error analysis and iterative refinement are disabled (ICNTL(11) and ICNTL(10))

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

If the complete system should be solved using the Schur complement matrix (see Subsection 3.19, ICNTL(26)=1 or 2), then the following parameters must be defined 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, sparse or distributed matrix (see Subsection 5.18).

mumps_par%**LREDRHS** is an optional integer parameter defined on the host to hold 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.

mumps_par%**REDRHS** is a **real** (**complex** in the complex version) one-dimensional pointer array that should be allocated on the host 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 (15)). 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 5.18).

5.20 Block Low-Rank (BLR) feature (ICNTL(35) and CNTL(7))

The current version of MUMPS implements the Block Low-Rank approach described in Subsection 3.20. In this section, we first give some recommendations at installation (Subsection 5.20.1), we then describe the Application Program Interface (API) to control the use of the BLR factorization (Subsection 5.20.2), and finally comment on the statistics that are provided on output (Subsection 5.20.3).

5.20.1 MUMPS installation and BLR functionality

Some precautions must be taken at installation to enable correct and efficient use of the BLR functionality.

External ordering: It is strongly advised to interface MUMPS with external ordering packages (Metis [41] or SCOTCH [49]) to enable the BLR representation of a frontal matrix to rely on K-way partitioning [57].

Multithreading: On multiprocessor nodes with a shared memory it is strongly advised to use multithreading and combine multithreaded BLAS with OpenMP directives-based multithreading (see Subsection 3.14).

MUMPS assumes that the BLAS library is compatible with OpenMP and both multithreaded BLAS and sequential BLAS in appropriate portions of the code will be used: serial BLAS when parallelism comes from OpenMP parallel regions and multithreaded BLAS when there is not enough parallelism from OpenMP. In this case, only OMP_NUM_THREADS should be set and the number of threads for the BLAS should not be fixed (e.g., with MKL do not set MKL_NUM_THREADS).

In case you use a multithreaded implementation of the BLAS routines that is not compatible with OpenMP and cannot adapt the number of threads to the current OpenMP region, then you can still use a multithreaded BLAS library, but you should compile MUMPS with the option -DBLR_NOOPENMP. This will lead to a performance loss with the BLR feature because multithreaded BLAS will be used on small matrices which would otherwise have been processed in parallel at a higher level with OpenMP.

Note that, if you mainly use BLR compression on large matrices another possibility is to use OpenMP with a serial BLAS library.

As these two options will lead to suboptimal performance, we recommend to use a BLAS library compatible with OpenMP when available.

LAPACK library: Please note that, low-rank approximations are computed using a truncated QR factorization with column pivoting, 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.

5.20.2 BLR API

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

ICNTL(35) controls the activation of the BLR feature

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

Possible values :

- 0 : Standard analysis and factorization (BLR feature is not activated).
- 1 : BLR feature is activated and automatic choice of BLR option is performed by the software.
- 2 : BLR feature is activated during both the factorization and solution phases, which allows for memory gains by storing the factors in low-rank.
- 3 : BLR feature is activated during the factorization phase but not the solution phase, which is still performed in full-rank. As a consequence, the full-rank factors must be kept and no memory gains can be obtained. In an OOC context, (ICNTL(22)=1) this option enables the user to write *all* factors to disk which is not the case with ICNTL(35)=2 since factors in low-rank form are not written to disk.

Other values are treated as 0.

Default value: 0 (standard multifrontal factorization).

Related parameters: CNTL(7) (BLR approximations accuracy), ICNTL(36) (BLR factorization variant), ICNTL(37) (compression of the contribution blocks), ICNTL(38) (estimation of the compression rate of the factors) and ICNTL(39) (estimation of the compression rate of the contribution blocks).

Incompatibility: Note that the activation of the BLR feature is currently incompatible with elemental matrices (ICNTL(5) = 1) (see error -800, subject to change in the future), and when the forward elimination during the factorization is requested (ICNTL(32) = 1), see error -43.

Remarks: If ICNTL(35)=1, then the automatic choice of BLR option is to activate BLR feature during both factorization and solution phases (ICNTL(35)=2). In order to activate the BLR factorization, ICNTL(35) must be equal to 1, 2 or 3 before the analysis, where some preprocessing on the graph of the matrix is needed to prepare the low-rank factorization. The value of ICNTL(35) can then be set to any of the above values on entry to the factorization (e.g., taking into account the values returned by the analysis). On the other hand, if ICNTL(35)=0 at analysis, only ICNTL(35)=0 is allowed for the factorization (full-rank factorization). When activating BLR, it is recommended to set ICNTL(35) to 1 or 2 rather than 3 to benefit from memory gains.

CNTL(7) is the dropping parameter ε (double precision real value) controlling the accuracy of the Block Low-Rank approximations.

Phase: accessed by the host during the factorization phase when ICNTL(35)=1, 2 or 3

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: The value of CNTL(7) 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 [23] _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 scale the matrix or or let the solver automatically preprocess (e.g., scale) the input matrix.

Note that, depending on the application, gains can be expected even with small values (close to machine precision) of CNTL(7).

ICNTL(36) controls the choice of the BLR factorization variant

Phase: accessed by the host during the factorization phase when ICNTL(35)=1, 2 or 3

Possible values :

0 : Standard UFSC variant with low-rank updates accumulation (LUA)

1 : UCFS variant with low-rank updates accumulation (LUA). This variant consists in performing the compression earlier in order to further reduce the number of operations and leads to a better tradeoff between flops reduction and accuracy.

Other values are treated as 0.

Default value: 1 (UCFS variant).

Related parameters: ICNTL(35) and CNTL(1)

Remarks: If numerical pivoting is not required and thus CNTL(1) can be set to 0.0, further performance gains can be expected with the UCFS version.

ICNTL(37) controls the compression of the contribution blocks (CB)

Phase: accessed by the host during the factorization phase when ICNTL(35)=1, 2 or 3

Possible values :

0 : contribution blocks are not compressed

1 : contribution blocks are compressed, reducing the memory consumption at the cost of some additional operations

Other values are treated as 0.

Default value: 0 (contribution blocks not compressed).

Related parameters: ICNTL(35), CNTL(7), ICNTL(40)

Remarks: This feature should be activated if memory consumption is a primary concern; note that it is likely to increase the factorization time. We recommend to combine this feature with ICNTL(40)=1 since BLR contribution blocks will then also benefit from adaptive precision representation for further storage reductions.

Estimating the memory footprint during analysis is difficult because the compression rate of the BLR factors is only known after factorization. (When ICNTL(35)=2, factors are kept in compressed form and the memory footprint can be reduced.) To enable the user to estimate during analysis the effect of BLR compression on the memory footprint, ICNTL(38) has been introduced. ICNTL(38) only influences the statistics printed during the analysis phase (only the INFO/INFOG arrays are affected). Furthermore, Out-Of-Core can also be combined with BLR compression. In this case, the factors that are kept full-rank (all of them if ICNTL(35)=3, or the ones of the frontal matrices not considered for BLR compression if ICNTL(35)=2) will be written onto the disk during the factorization. BLR estimations of the peak of memory for all possible situations are provided during analysis.

After factorization, statistics on the effective low-rank compression and on the memory effectively allocated/used are then provided to the user (see Section 5.20.3) and can then be used to adjust the value of ICNTL(38).

Similarly, with ICNTL(39) the user can provide an estimation of the compression rate of the contributions blocks.

ICNTL(38) estimated compression rate of LU factors

Phase: accessed by the host during the analysis and the factorization phases when ICNTL(35)=1, 2 or 3

Possible values : between 0 and 1000 (1000 is no compression and 0 is full compression); other values are treated as 0; ICNTL(38)/10 is a percentage representing the typical compression of the factor matrices in BLR fronts: $\texttt{ICNTL(38)}/10 = \frac{\text{compressed factors}}{\text{uncompressed factors}} \times 100.$

Default value: 600 (when factors of BLR fronts are compressed, their size is 60.0% of their full-rank size).

Related parameters: ICNTL(35), CNTL(7)

Remarks: Influences statistics provided in INFO(29), INFO(30), INFO(31), INFOG(36), INFOG(37), INFOG(38), INFOG(39) , but also INFO(32-35) and INFOG(40-43)

ICNTL(39) estimated compression rate of contribution blocks

Phase: accessed by the host during the analysis and the factorization phases when ICNTL(35)=1, 2 or 3, and ICNTL(37)= 1

Possible values : between 0 and 1000 (1000 is no compression and 0 is full compression); other values are treated as 0; ICNTL(39)/10 is a percentage representing the typical compression of the CB in BLR fronts: ICNTL(39)/10 $= \frac{\text{compressed CB}}{\text{uncompressed CB}} \times 100$.

Default value: 500 (when CB of BLR fronts are compressed, their size is 50% of their full-rank size).

Related parameters: ICNTL(35), ICNTL(37)

Remarks: Influences statistics provided in INFO(32), INFO(33), INFO(34), INFO(35), INFO(36), INFO(37), INFO(38), INFOG(40), INFOG(41), INFOG(42), INFOG(43) INFOG(44) INFOG(45) INFOG(46) INFOG(47)

5.20.3 BLR output: statistics

In the following, output statistics produced after the BLR factorization (see Section 3.20) are indicated. The effect of BLR compression on both the number of operations and the number of entries in the factors is reported.

Please note that when ICNTL(35)=2, factors are stored in compressed form and will be used in compressed form during the solve phase. The effective size used to store the factors (INFOG(9)) thus depends on the fact that ICNTL(35) is set to 2. The number of operations during the solve phase is related to the size of the factors and will also benefit from the compression.

```
-------------- Beginning of BLR statistics --------------------------------
 ICNTL(36) BLR variant                          = 1
 CNTL(7)   Dropping parameter controlling accuracy = 1.0E-06
 Statistics after BLR factorization
      Number of BLR fronts              = 847
      Fraction of factors in BLR fronts = 90.8%
      Statistics on the number of entries in factors:
      INFOG(29) Theoretical nb of entries in factors    = 4.881E+06 (100.0%)
      INFOG(35) Effective nb of entries (% of INFOG(29))= 1.599E+06 ( 32.7%)
      Statistics on operation counts (OPC):
      RINFOG(3) Total theoretical operations            = 1.985E+09 (100.0%)
      RINFOG(14) Total effective OPC   (% of RINFOG(3)) = 2.243E+08 ( 11.3%)
 ---------- End of BLR statistics ---------------------------------------
```

This output reports the following information:

- Number of BLR fronts: not all frontal matrices are considered for compression. Only those that fulfill a minimum size requirement are considered.
- Fraction of factors in BLR fronts: corresponds to the fraction of full-rank factors that will be be considered for compression (i.e. the ratio of the number of entries in the factors of all frontal matrices on which BLR compression will be performed over the total number of entries (INFOG(29)).
- INFOG(29) Theoretical nb of entries in factors: number of entries in the factors (sum over all processors), if a standard (full-rank) factorization had been performed.
- INFOG(35) Effective nb of entries: effective number of entries in the factors (sum over all processors) taking into account BLR factor compression. The value reported in parentheses is the fraction of the theoretical number of entries in full-rank factors INFOG(29).
- RINFOG(3) Total theoretical full-rank OPC: number of (real or complex) floating-point operations (sum over all processors), if a standard standard (full-rank) factorization had been performed. It is provided here as a reference.
- RINFOG(14) Total effective OPC: actual number of floating-point operations done by the factorization phase with BLR feature. The value reported in parentheses is the fraction of the theoretical full-rank operation count RINFOG(3). Using the BLR feature, the reduction in time for factorization should be related to the percentage reduction of full-rank operations.

5.20.4 Adaptive precision BLR

As described in Subsection 3.20.3, an adaptive precision BLR feature is implemented in MUMPS. It can be activated with the parameter ICNTL(40) described below. On CPUs supporting the AVX512-VBMI instruction set, the conversion from custom floating-point types to standard ones can exploit vectorization for optimal performance. For this feature to be enabled, the flag -DUSE_AVX512_VBMI that should be added to the OPTF and OPTC variables of the Makefile.inc used to compile MUMPS. In order for your compiler to compile for this instruction set, you should also add -march=native or -mavx512vbmi -mavx512f to the OPTC variable of the Makefile.inc used to compile MUMPS (see GNU march description, or refer to your compiler documentation). Please also refer to WikiChip, lscpu, or your CPU vendor for available instruction set on your CPU.

ICNTL(40) activates adaptive precision Block Low-Rank feature

Phase: accessed by the host during factorization phase when ICNTL(35) =1, 2 or 3

Possible values :

0 : no adaptive precision BLR

1 : adaptive precision BLR to reduce the memory footprint of the BLR LU factors.

Default value: 0 (no adaptive precision BLR)

Related parameters: ICNTL(35), CNTL(7) and ICNTL(37)

Remarks: In this version, the use of adaptive precision mainly targets storage reductions. The reduced size of the BLR LU factors can, however, also accelerate the triangular solve phase in some cases. With adaptive precision BLR activated on should expect to reduce INFOG(9) since the space needed to *store* the BLR factors can be further reduced.

5.21 Save (JOB=7) / Restore (JOB=8) feature

To save to disk MUMPS internal data associated to a given instance, MUMPS should be called with JOB= 7 (see Subsection 5.1.1). These MUMPS internal data are saved in binary files. It is possible to use the Save feature (JOB=7) before or after any of the main phases (analysis, factorization, solve, JOB=1,2,3,4,5,6).

After that, it is possible to continue working with the existing instance until, at some point, the instance should be terminated (JOB= –2). In order to restart MUMPS with the saved data, the user should first create a new instance (JOB= –1, see Subsection 5.1.1) and then restore into that instance the saved data with a call to MUMPS with JOB= 8. Note that arrays that are allocated and freed by the user are not saved (JOB= 7) nor restored (JOB= 8), although some of them might be requested for further calls to MUMPS. For example, the arrays associated to the input matrix are not saved. See important remarks on how to use this feature in Subsection 5.21.3.

5.21.1 Location and names of the save files

The save files are written in the directory provided by the user. Their names may start with an user defined prefix. The following variables are involved:

mumps_par%**SAVE_DIR** (string, maximum length of 1023 characters) must be provided by the user (on each processor) to control the directory where MUMPS internal data will be stored.

It is also possible to provide the directory through environment variables. If SAVE_DIR is not defined, then MUMPS checks for the environment variable MUMPS_SAVE_DIR. If neither SAVE_DIR nor the environment variable MUMPS_SAVE_DIR are defined, then error -77 will be raised.

mumps_par%**SAVE_PREFIX** (string, maximum length of 255 characters)

can be provided by the user (on each processor) to prefix the files. It is also possible to provide the file prefix through environment variables. If SAVE_PREFIX is not defined, then MUMPS checks for the environment variable MUMPS_SAVE_PREFIX. If neither SAVE_PREFIX nor the environment variable MUMPS_SAVE_PREFIX are defined, then MUMPS will use save.

To compound the file names, the prefix strings (eventually defined with SAVE_PREFIX) will be appended at least with *_myid* in case of MPI, where *myid* is the MPI rank of the current processor. The file name extension is .mumps, so the files can easily be detected afterwards.

A text file containing a recap of the saved instance is also created with the same basename, on every processor. The file name extension is .info and the files are meant to be read by the user.

5.21.2 Deletion of the save files (JOB=-3)

Once the binary files are not needed anymore, i.e. no other restart is planned with these data, it is possible to delete the files by calling MUMPS with JOB= –3. MUMPS will access the structure components SAVE_DIR and SAVE_PREFIX to know which files should be deleted. If the user does not specify the directory in the structure components mumps_par%SAVE_DIR and/or the prefix of the files in mumps_par%SAVE_PREFIX, MUMPS will check for the environment variable MUMPS_SAVE_DIR for the directory, and for the environment variable MUMPS_SAVE_PREFIX or use save for the prefix name.

During the deletion phase (JOB= –3), MUMPS does not restore the files in SAVE_DIR with SAVE_PREFIX in the current MUMPS instance. Thus it is possible to remove some previously saved files and to pursue the computations with the current instance.

In case of out-of-core (ICNTL(22)=1), special care should be taken to manage associated out-of-core files. See Subsection 5.21.4 for more details.

5.21.3 Important remarks for the restore feature (JOB=8)

A call to MUMPS with JOB= 8 must be preceded by a call with JOB= –1 on the same instance.

Requested parameters: the parameters COMM, SYM and PAR are not restored and must be given back by the user before the call to MUMPS with JOB= –1. The values SYM and PAR must be identical to the values of the saved instance. Also, the size of the communicator COMM should be the same as the size of the one that was used at the moment of the save (JOB= 7).

Control parameters: during the restoration of MUMPS data, the values of ICNTL and CNTL are set back to the saved values. So any intended change of these values should occur **after** the call of MUMPS with JOB= 8 to be taken into account during the following calls of MUMPS.

Constraint with binary files: because binary files are used to save the data, the binary storage should have identical format (for example, little or big endian) on the machine that saves the data and the one that restores it.

Arrays allocated and freed by the user As mentioned above, the arrays that are allocated and freed by the user are not saved and cannot be restored by this feature, although some of them might be requested for further calls to MUMPS. For example, the arrays associated to the input matrix are not stored. The user is thus responsible of providing them again, if they are going to be needed. Data from the user that are not saved but that might be required internally by MUMPS after the restoration include:

- **Input matrix:** the arrays associated to the input matrix (see Subsection 5.4.2) are not saved during a call of MUMPS with JOB= 7. The possible arrays concerned are: IRN, JCN, A, IRN_loc, JCN_loc, A_loc, ELTPTR, ELTVAR, A_ELT. Consequently, the same matrix data must be provided again by the user before a (possibly new) factorization or before a solve if iterative refinement (ICNTL(10)) or error analysis (ICNTL(11)) are requested.
- **Scaling arrays:** the scaling arrays ROWSCA and COLSCA are not saved by the save phase in case they had been provided by the user before the factorization and should then be provided again before the solve phase.
- **Right-hand side vector:** the Right-hand side (or Solution) vector RHS is not saved by the save phase.

MPI context: the only constraint is that before a call to MUMPS with JOB= 8 using the communicator COMM provided in the JOB= –1 call, the process with rank<*myid*> should have access to the file <SAVE_DIR>/<SAVE_PREFIX>*_myid*.mumps. There are 6 possibilities :

<SAVE_DIR>/<SAVE_PREFIX>*_myid*.mumps

<SAVE_DIR>/<MUMPS_SAVE_PREFIX>_*myid*.mumps
<SAVE_DIR>/save_*myid*.mumps
< MUMPS_SAVE_DIR>/<SAVE_PREFIX>_*myid*.mumps
< MUMPS_SAVE_DIR>/<MUMPS_SAVE_PREFIX>_*myid*.mumps
< MUMPS_SAVE_DIR>/save_*myid*.mumps

5.21.4 Combining the Save/Restore feature with out-of-core

The Save/Restore feature is fully compatible with the out-of-core computations (ICNTL(22) = 1). After a call to MUMPS with JOB= 7, the files storing the factors are associated to a saved instance. They will be needed if the saved instance is restored. Thus they are not deleted, neither during the termination phase (JOB= –2), nor at the beginning of a new factorization (JOB= 2), unlike what usually happens with out-of-core files (see Subsection 5.11).

After a save (JOB= 7) or a restore (JOB= 8), it is still possible to perform a new factorization (JOB= 2) using the out-of-core feature. In this case, new out-of-core files are produced. The old out-of-core files are not deleted since they are associated to a saved instance. On the contrary, the new files storing the new factors are not associated to a saved instance. They will be treated as usual out-of-cores files, i.e. they will be deleted during the termination phase (JOB= –2) or at the beginning of an other new factorization.

When MUMPS is called with JOB= –3, the out-of-core factor files associated to the saved instance are marked out for deletion. This means the current out-of-core files can be used again during solve phases. But because they are not associated to a saved instance, they will be deleted during the termination phase or at the beginning of a new factorization.

In the lifetime of a classical MUMPS instance, the out-of-core files are kept up to the termination phase or a new factorization. If several saves (JOB= 7) are performed after the same out-of-core factorization, they will all refer to the same out-of-core files. In such a situation, the user might need advanced controls on the out-of-core files management available using ICNTL(34).

ICNTL(34) controls the conservation of the OOC files during JOB= –3.

Phase: accessed by the host during the Save/Restore files deletion phase (JOB= –3) in case of out-of-core (ICNTL(22)=1).

Possible values :

0: the out-of-core files are marked out for deletion

1: the out-of-core files should not be deleted because another saved instance references them.

Other values are treated as 0.

Default value: 0 (out-of-core files associated to a saved instance are marked out for deletion at the end of the out-of-core file lifetime)

Remarks: MUMPS will delete only the out-of-core files that are referenced in the saved data identified by the value of SAVE_DIR and SAVE_PREFIX. Extra out-of-core files with the same OOC_TMPDIR and OOC_PREFIX are not deleted.

5.21.5 Combining the Save/Restore feature with WK_USER

In case an optional workspace is provided by the user and a call to MUMPS with Save feature (JOB= 7) is executed, MUMPS will not save either restore LWK_USER and WK_USER.

During the creation of a new instance (JOB= –1) called after a Save, LWK_USER is set to zero and WK_USER pointer is nullified. After the call to MUMPS with JOB= –1 and if a solve step without factorization (JOB= 3) is requested, MUMPS will expect the user to provide access to a new workspace WK_USER of size LWK_USER. The user should then load in this new workspace the data returned in WK_USER by MUMPS during the factorization step performed before Save (JOB= 7). MUMPS will then check that the new workspace provided is large enough *or has the same size as the one provided in an earlier call*. If not, MUMPS will raise an error with the error code −11.

5.21.6 Error management

In case of error while saving or restoring an instance, an error code will be returned in INFO(1) and INFOG(1) (see Section 7) and no files are created.

In case the user wants to Save (JOB= 7)) an instance after a call to MUMPS that returned with an error (INFO(1)<0 and INFOG(1)<0), this is possible, although it is not recommended. After the save, the values of INFO(1), INFO(2), INFOG(1), INFOG(2) on exit will represent the success or failure of the save phase. Similarly, after restoring a saved instance (JOB= 8), INFO(1), INFO(2), INFOG(1), INFOG(2) are associated to the success or failure of the restore operation, not of the last call to MUMPS done before saving the instance.

It is still possible to pursue the computations on the restored instance after the last successful step. For example, one can perform a factorization (JOB= 2) on a restored instance for which the analysis (JOB= 1) was successful but the factorization failed before saving the instance.

5.22 Setting the number of OpenMP threads by MUMPS (ICNTL(16))

For normal and safe setting of multithreaded parallelism (in combination with MPI or not) please refer to Subsection 3.14.

When, for some reasons, setting the number of OpenMP threads is not possible, then one can ask MUMPS to set the number of OpenMP threads. This is done internally by a call to the OpenMP function omp_set_num_threads. If this possibility is activated, during the whole current call to MUMPS, ICNTL(16) OpenMP threads will be used. On exit, the number of OpenMP threads is set back to the number of OpenMP threads before the MUMPS call.

ICNTL(16) controls the setting of the number of OpenMP threads by MUMPS and should be used only when the setting of multithreaded parallelism is not possible outside MUMPS (see Subsection 3.14)

Phase: accessed by the host at the beginning of all phases

Possible values :

- 0 : (recommended value) nothing is done, MUMPS uses the number of OpenMP threads configured by the calling application.
- > 0 : on entry MUMPS sets the number of OpenMP threads to the value of ICNTL(16) and on exit reset the number of OpenMP threads to its value before the MUMPS call.

Other values are treated as 0.

Default value: 0 (nothing is done)

Remarks: Please note that the use of ICNTL(16) should be limited to the case when for some reasons, setting the number of OpenMP threads is not possible outside MUMPS.

5.23 Converting MPI processes to OpenMP threads (ICNTL(17))

This feature enables MUMPS to use some MPI processes as OpenMP resources during the numerical phases. It is assumed that MUMPS is compiled with OpenMP (see Section 3.14). For this feature to be enabled, the flag -DMPI_TO_K_OMP that should be added to OPTF and OPTC variables of the Makefile.inc used to compile MUMPS. The appropriate OpenMP compiler and linker options should be used as described in Subsection 3.14.

ICNTL(17) controls the *MPI to k OMP* grouping factor to reuse MPI processes as OpenMP resources

Phase: accessed by the host at the beginning of the analysis phase

Possible values :

- 0 : feature not activated
- $k \geq 2$: value of the *MPI to k OMP* reuse factor. The number of MPI processes will be divided by k during computations in the factorization and solve phases while the number of OpenMP threads per MPI process will be multiplied by k
- -1 : Automatic setting of *MPI to k OMP* reuse factor

$k < -1$: Try to use up to (but not more than) $|k|$ threads per MPI process (taking into account threads already "attached" to MPI processes)

Default value: 0 (functionality not used)

Remarks:

- This feature has been tested only on Linux platforms.
- To use the *MPI to k OMP* and take the most advantage of it, a way to manage the affinity masks must be provided. We propose two portable solutions, based on a C low-level system API from `sched.h`, and on the external library HWLOC[9], respectively.
  - To use the C low-level system API (`sched.h`), please provide the option `-DUSE_SCHEDAFFINITY` in the `OPTF` and `OPTC` variables of the Makefile.inc used to compile `MUMPS`.
  - To use the external library HWLOC, please compile `MUMPS` with `-DUSE_LIBHWLOC` (to be also provided in `OPTF` and `OPTC` of the Makefile.inc), add `-lhwloc` (for example in `LIBOTHERS` in `MUMPS` Makefile), add the path to the library and to the include directory if needed.

  If both `-DUSE_LIBHWLOC` and `-DUSE_SCHEDAFFINITY` are used when building `MUMPS`, HWLOC will be used.
- Let us suppose that the number of MPI processes per node is 18 and that `NOMP`=2 where `NOMP` holds the number of OpenMP threads per MPI process (`NOMP=omp_get_max_threads()`). Examples of values for `ICNTL(17)`:
  - 6 : the effective value of the grouping factor is 6, leading to 12 threads per MPI process and 3 working MPI processes per node
  - 5 : error return `INFO(1)`=`-60` because 5 does not divide 18
  - -9: the effective value of the grouping factor is 3 (3 is the largest possible value that divides 18 and that is such that $3 * NOMP \leq 9$), leading to 6 threads per MPI process and 6 working MPI processes per node
- For performance analysis and debugging the user may want to check how MPI processes involved in the numerical phases (which depends on `PAR`) are grouped, the original affinity masks of the MPI processes, and the possibly modified affinity masks of the MPI active workers. The grouping of the MPI processes is printed as a global statistic (unit `ICNTL(3)` when `ICNTL(4)` ≥ 2). In order to obtain more information on the affinity masks, the preprocessing option `-DMTKO_VERBOSE` should be provided in the `OPTF` and `OPTC` variables of the file Makefile.inc used to compile `MUMPS`. Additional traces starting with `"MPI TO K OMP"` will then be printed when `ICNTL(2)` > 0 and `ICNTL(4)` ≥ 2.

5.24 Compact workarray `id%S` at the end of factorization phase

When memory for the solve phase is critical (case of large number of right-hand sides) or when the memory footprint at the end of the factorization phase needs to be reduced, setting `ICNTL(49)` to 1 or 2 enables to compact the internal workarray `id%S` at the end of the factorization so that `id%S` only holds factor matrices needed for the solution phase.

ICNTL(49) compact workarray `id%S` at the end of factorization phase

Phase: accessed by the host during factorization phase

Possible values :

- 0 : nothing is done.
- 1 : compact workarray `id%S(MAXS)` at the end of the factorization phase while satisfying the memory constraint that might have been provided with `ICNTL(23)` feature.
- 2 : compact workarray `id%S(MAXS)` at the end of the factorization phase. The memory constraint that might have been provided with `ICNTL(23)` feature does not apply to this process.

[9] https://www.open-mpi.org/software/hwloc

Other values are treated as 0.

Default value: 0

Incompatibility: With the use of LWK_USER / WK_USER feature.

Remarks: ICNTL(49)=1,2 might require intermediate memory allocation to reallocate id%S of minimal size. If the memory allocation fails, then a warning is returned and nothing is done. If ICNTL(49)=1 and the memory constraint provided with ICNTL(23) $>$ 0 would not be satisfied then a warning is raised and nothing is done.

5.25 Stop on user request and progress information

.

MUMPS provides a mechanism to cancel computations when the user sets, during numerical computation phases, the control parameter ICNTL(50) to 1.

ICNTL(50) cancels computations in the middle of factorization or solve phase

Phase: accessed by working MPI processes during the factorization and solve phases

Possible values :

0 : nothing is done, MUMPS performs its computations normally.

1 : user wants to cancel the MUMPS execution. When a process detects that ICNTL(50) has been set to 1, it informs the other processes and the computations are cancelled even if the factorization (JOB= 2) or the solve (JOB= 3) are not finished. The error code returned is INFO(1)=−80 if MUMPS was cancelled during the factorization phase, and INFO(1)=−81 if MUMPS was cancelled during the solve phase. INFO(2) holds the rank of the MPI process on which ICNTL(50) was set to 1.

Other values are treated as 0.

Default value: 0

Remarks: In case of non-working host (PAR=0), ICNTL(50) should be set on a working MPI process, i.e. on a process with rank greater than 0 in the communicator COMM passed to MUMPS. Furthermore, if the MUMPS execution was cancelled with ICNTL(50)=1, it is the user's responsibility to reset ICNTL(50) to 0 before a new call to MUMPS.

MUMPS also provides progress information during the factorization phase: the theoretical number of operations performed by each MPI process can be obtained by accessing the value of RINFO(3). In case of multiple MPI processes, RINFO(3) is local to each MPI process. Still, comparing RINFO(3) to the estimated flops RINFO(1) on a given MPI process provides an indication of the work performed with respect to the overall amount of work on that process. Note that, if RINFO(3) becomes greater than RINFO(1), this is typically because the number of floating-point operations has increased due to numerical pivoting.

5.26 Improved multithreading using tree parallelism (ICNTL(48)), so called $\mathcal{L}_0$-threads feature

In a shared memory context, or in an hybrid distributed-shared memory context, multithreading used to be exploited at the node parallelism level only, i.e., different fronts were not factored concurrently by different threads. However, in multifrontal methods, multithreading may exploit both node and tree parallelism. Such an approach has been proposed during the PhD thesis of Wissam Sid Lakhdar [54, 45] and relies on the idea of separating the fronts by a so-called $\mathcal{L}_0$-threads layer, as illustrated in Figure 4.

Each subtree rooted at the $\mathcal{L}_0$ layer is treated sequentially by a single thread; therefore, below the $\mathcal{L}_0$ layer pure tree parallelism is exploited by using all the available threads to process multiple sequential subtrees concurrently. When all the sequential subtrees have been processed, the approach reverts to pure node parallelism: all the fronts above the $\mathcal{L}_0$ layer are processed sequentially (i.e., one after the other) but all the available threads are used to assemble and factorize each one of them. The research code

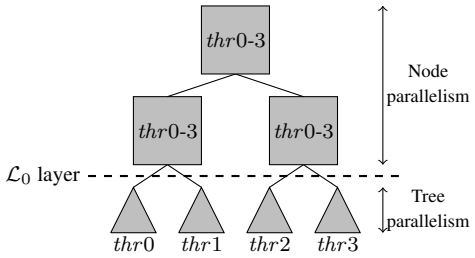


Figure 4: Simplified illustration with four threads of how both node and tree multithreading can be exploited.

developed in the context of the PhD thesis [54] has also been used to improve the parallel behaviour of the BLR approach in a multithreaded environment [10].

The proposed new feature includes the work developed during the thesis and has been extended on the one hand to be more robust and and on the other hand to exploit tree parallelism also during the solve phase.

As a result, the multithreaded performance of all the numerical phases (factorization and solve) should be impacted. The gain in performance should be particularly significant on large number of threads to solve problems corresponding to 2D and 2.5D grid problems.

In case of MPI parallelism with multiple threads per MPI process, it is also applied to improve the performance within each MPI process.

ICNTL(48) Multithreading with tree parallelism

Phase: accessed by the host during the analysis phase (need be set after the initialization phase (JOB= −1)) and during the solution phase

Possible values :

0 : not activated

1 : multithreaded tree parallelism activated

Other values are treated as 0.

Default value: 1

Related parameters: if tree parallelism is activated, then the number of threads per MPI process set through the OMP_NUM_THREADS environment variable or ICNTL(16) control parameter should be at least 2. It will influence the choice of the $\mathcal{L}_0$ layer at the analysis phase and should not be modified during the subsequent numerical phases.

Remarks:

- Please note that, once ICNTL(48) $= 1$ is set prior to the analysis phase it will be used for both analysis and factorization phases. Tree parallelism can be switched off (ICNTL(48)=0) prior to the solution phase (JOB= 3).
- The number of threads per MPI process should be identical and if not multithreaded tree parallelism will be switched off during analysis.
- If tree parallelism is not effective during analysis then it cannot be activated during the factorization and solve phases.
- The memory provided with WK_USER will not be used under the $\mathcal{L}_0$ layer making WK_USER not recommended when ICNTL(48) $= 1$.

5.27 Offload activities on GPUs (ICNTL(51))

In the last few years, nodes of parallel computers have become very heterogeneous, leading to extremely powerful nodes equipped with multicore CPUs together with accelerators such as GPUs.

This version includes a layer to replace some of the calls to the BLAS (Basic Linear Algebra Subprograms) library by calls to cuBLASXT or XKBlas [34] routines. Both the cuBLASXT and

XKBlas libraries are designed on top of the cuBLAS library (an implementation of BLAS on top of the NVIDIA®CUDA™ runtime) and take care of the data transfer and allocation on the GPU needed to execute kernels of the cuBLAS library[10].

Memory pinning is useful to reduce the cost of data transfers between CPUs and GPUs but we observed pinning operations to have a significant cost. To limit this extra cost, we have implemented an asynchronous pinning strategy, where a thread is in charge of all pinning and unpinning requests. Memory is pinned at the beginning of the assembly of a frontal matrix and is unpinned at the end of the factorization. Waiting the end of pinning/unpinning requests is performed before the GPU data transfer. Under the $\mathcal{L}_0$-threads layer where each front is processed by only one thread and many threads can simultaneously offload activities on GPUs, we have observed that it is currently preferable not to pin memory.

Whereas in previous versions, GPU offloading mainly targeted executions with a single MPI process, this version has been significantly improved in MPI environments to offload operations associated to distributed frontal matrices and stabilized previous experimental approaches. It uses GPUs for BLAS-3 calls on the MPI processes responsible of updating the non fully-summed rows of distributed frontal matrices but also on the MPI process responsible of factorizing the fully-summed rows of such distributed frontal matrices, leading to a large portion of work on GPU. BLAS operations on distributed frontal matrices are now compatible with synchronous and asynchronous pinning strategies, thanks to the management of memory states and the possibility to asynchronously unpin and pin data in case chunks of data are moved. The granularity of operations has been increased to increase the arithmetic intensity and the efficiency of GPU operations with respect to data transfers (by setting ICNTL(51) > 0 at analysis). Finally, when using XKBlas, less data transfers are possible thanks to its natural asynchronous approach. For example, if the output of a TRSM operation is input to a GEMM operation, data can stay on GPU (depending on available memory on GPU).

To better analyse/understand performance we provide after factorization the amount of flops (RINFO(9)/RINFOG(22)) as well as the time (RINFO(10)/RINFOG(23)) spent on GPUs.

INSTALLATION AND PARAMETER SETTING:

- -DUSE_GPU should be added to OPTF and OPTC in the file Makefile.inc of MUMPS.
- -I${CUDA_HOME}/include should be added to OPTC
- -L${CUDA_HOME}/lib64 -lcublas -lcudart to OPTL
- To use/install XKBlas (CeCILL-C free license), -DUSE_XKBLAS should be added to OPTF and OPTC of the file Makefile.inc of MUMPS. Furthermore, -lxkblas -lkaapi should be added to OPTL.

ICNTL(51) GPU activation: number of GPUs per MPI process

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

Possible values :

– during analysis:

≤ 0 : nothing done

$k > 0$: modify analysis (increase granularity and modify scheduling of tasks, increase buffer sizes) to expose more work to GPUs.

– during factorization:

≤ 0 : do not use GPU acceleration

$k > 0$: use k GPUs per MPI process

Default value: 0

Remarks:

– $k = 1$ is recommended; k should be smaller or equal to the number of GPUs accessible by each MPI process.

– In case of multiple MPI processes, only the value of ICNTL(51) on the host (MPI process with rank 0) is meaningful and all the MPI processes are assumed to use the same number of GPUs.

[10]The authors of XKBlas are now also addressing AMD GPUs.

– In case of multiple MPI processes per node and $k > 1$, it may happen that a GPU will be shared by several MPI processes.

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 60.

- 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(15) exploits compression of the input matrix resulting from a block format
- ICNTL(16) controls the setting of the number of OpenMP threads
- ICNTL(17) controls the *MPI to k OMP* reuse factor value
- ICNTL(18) defines the strategy for the distributed input matrix
- ICNTL(19) computes the Schur complement matrix
- ICNTL(20) determines the format (dense, sparse, or distributed) 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 process
- 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 has been computed
- 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(34) controls the deletion of the files in case of Save/Restore
- ICNTL(35) controls the activation of the Block Low-Rank (BLR) feature
- ICNTL(36) controls the choice of BLR factorization variant
- ICNTL(37) controls the BLR compression of the contribution blocks
- ICNTL(38) estimates compression rate of LU factors
- ICNTL(39) estimates compression rate of contribution blocks
- ICNTL(40) activates adaptive precision BLR feature
- ICNTL(41-46) reserved in current version
- ICNTL(47) controls the possibility to perform a single precision factorization in a double precision MUMPS instance
- ICNTL(48) controls multithreading with tree parallelism
- ICNTL(49) compact workarray `id%S` at the end of factorization phase
- ICNTL(50) cancels `MUMPS` execution on user request
- ICNTL(51) Offload activities on GPUs
- ICNTL(52-55) reserved in current version
- ICNTL(56) detects pseudo-singularities during factorization and factorizes the root node with a rank-revealing method
- ICNTL(57) reserved in current version
- ICNTL(58) defines options for symbolic factorization
- ICNTL(59-60) not used in current version

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 and statistics local to each MPI process.

Possible values :

≤ 0: these messages will be suppressed.
> 0 : is the output stream.

Default value: 0

Remarks: If `ICNTL(2)` > 0 and `ICNTL(4)` ≥ 2, then information on advancement (flops done) is also printed.

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, warnings and main statistics printed)

ICNTL(5) controls the matrix input format (see Subsection 5.4.2).

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 5.4.2.1) or in the structure components N, NNZ_loc (or NZ_loc), IRN_loc, JCN_loc, A_loc if the matrix is distributed (see Subsection 5.4.2.2).

1 : elemental format. The matrix must be input in the structure components N, NELT, ELTPTR, ELTVAR, and A_ELT (see Subsection 5.4.2.3).

Any other values will be treated as 0.

Default value: 0 (assembled format)

Related parameters: `ICNTL(18)`

Incompatibility: If the matrix is in elemental format (`ICNTL(5)`=1), the BLR feature (`ICNTL(35)` ≥ 1) is currently not available, see error `-800`.

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 5.5.2).

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 more costly (time and memory footprint) 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 if SYM= 0 or 2 (automatic choice done by the package),

0 if SYM= 1 (no column permutation is computed)

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 [28, 29]) 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 5.5.1). Otherwise, the pointer is not associated. The column permutation is such that entry $a_{i,uns_perm(i)}$ 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 [30] 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 5.6.

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) [11] 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[11] [49] package is used if previously installed by the user otherwise treated as 7.

4 : PORD[12] [53] is used if previously installed by the user otherwise treated as 7.

5 : the Metis[13] [41] package is used if previously installed by the user otherwise treated as 7.
It is possible to modify some components of the internal options array of Metis (see Metis manual) in order to fine-tune and modify various aspects of the internal algorithms used by Metis. This can be done by setting some elements (see the file metis.h in the Metis installation to check the position of each option in the array) of the MUMPS array mumps_par%**METIS_OPTIONS** after the MUMPS initialization phase (JOB= –1) and before the analysis phase. Note that the METIS_OPTIONS array of the MUMPS structure is of size 40, which is large enough for both Metis 4.x and Metis 5.x versions. It is passed by MUMPS as the argument "options" to the METIS ordering routine METIS_NodeND (METIS_NodeWND is sometimes also called in case MUMPS was installed with Metis 4.x) during the analysis phase.

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 5.6.1.

ICNTL(8) describes the scaling strategy (see Subsection 5.5).

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

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

Possible variables/arrays involved: COLSCA, ROWSCA

Possible values :

-2: Scaling done during analysis (see [28, 29] for the unsymmetric case and [30] for the symmetric case), under certain conditions. The original matrix has to be centralized (ICNTL(18)=0) and the user has to provide its numerical values (mumps_par%A) on entry to the analysis, otherwise the scaling will not be computed. Also, ICNTL(6) must be activated for the scaling to be computed.
The effective value of the scaling applied is reported in INFOG(33). We recommend that the user checks for INFOG(33) in order to know if the scaling was applied. When not applied, other scalings can be performed during the factorization.

-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 [52, 20, 43, 42]) 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

Other values are treated as 77.

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 real and symmetric with SYM= 1 then the automatic choice for scaling is no scaling except when BLR feature (ICNTL(35)) is activated since scaling is recommended previous to BLR factorization.

Incompatibility: 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 input matrix is assembled and distributed (ICNTL(18)=1,2,3 and ICNTL(5) = 0), then only options 7, 8 and 77 are allowed, otherwise no scaling is applied.
If block format is exploited (ICNTL(15) $\neq$ 0) then scaling is not 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 5.8).

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), or if ICNTL(20)=10 or 11 (distributed right hand sides), then iterative refinement is disabled and 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 [24] 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 criterion (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 criterion (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 2)
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 5.9.

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 ICNTL(21)=1 (solution kept distributed) or if ICNTL(32)=1 (forward elimination during factorization), or if NRHS>1 (multiple right hand sides), or if ICNTL(20)=10 or 11 (distributed right hand sides), or if ICNTL(25)=-1 (computation of the null space basis), 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:

- 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 [30] 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 1.

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, or the analysis is performed by blocks (ICNTL(15) $\neq$ 0), 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 :

< -1 : treated as 0.
-1 : force splitting of the root node in all cases (even sequentially)

0 : parallel factorization of the root node based on ScaLAPACK. 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 : ScaLAPACK is not used (recommended value is 1 to partly recover parallelism of the root node). It forces a sequential factorization of the root node (ScaLAPACK will not be used). To recover parallelism lost by the fact of not using ScaLAPACK, splitting of the root node can be activated: if the number of working processes is strictly larger than ICNTL(13) (always the case with ICNTL(13)=1) then splitting of the root node is performed to enable node level parallelism.

Default value: 0 (parallel factorization on the root node) except in the following situations. Default is 1 when offloading activities on GPU is requested (ICNTL(51)>0). With rank-revealing factorization (ICNTL(56)=1) the root is processed sequentially and ICNTL(13) setting is ignored.

Related parameters: ICNTL(51), ICNTL(56)

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 5.13) or to activate BLR compression (Subsection 5.20) on the root node.

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.

With SYM=1, if ScaLAPACK is allowed (ICNTL(13)$\leq$ 0) then Cholesky factorization will be performed on the root node and thus negative pivots will raise an error (code -40 is returned in INFOG(1)).

ICNTL(14) controls the percentage increase in the estimated working space, see Subsection 5.12.

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

Default value: between 20 and 35 (which corresponds to at most 35 % increase) and depends on the number of MPI processes. It is set to 5 % with SYM=1 and one MPI process.

Related parameters: ICNTL(23)

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

ICNTL(15) exploits compression of the input matrix resulting from a block format, see Subsection 5.7.

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

Possible variables/arrays involved: NBLK, BLKPTR, BLKVAR

Possible values :

0: no compression

-k: all blocks are of fixed size k> 0. N (the order of the matrix A) must be a multiple of k. NBLK and BLKPTR should not be provided by the user and will be computed internally. Concerning BLKVAR, please refer to the *Remarks* below.

1: block format provided by the user. NBLK need be provided on the host by the user and holds the number of blocks. BLKPTR(1:NBLK+1) must be provided by the user on the host. Concerning BLKVAR, please refer to the *Remarks* below.

Any other values will be treated as 0.

Default value: 0

Remarks: If BLKVAR is not provided by the user then BLKVAR is internally treated as the identity (BLKVAR(i)=i, (i=1, ..., N)). It corresponds to contiguous variables in blocks.

- If ICNTL(15)=1 then BLKVAR(BLKPTR(iblk):BLKPTR(iblk+1)-1), (iblk=1, NBLK) holds the variables associated to block iblk.
- If ICNTL(15) < 0 then BLKPTR need not be provided by the user and NBLK $= N/k$ where N must be a multiple of k.

In case the pivot order is provided on entry by the user at the analysis phase (ICNTL(7) $= 1$) then PERM_IN should be compatible with the compression. This means that PERM_IN, of size N, should result from an expansion of a pivot order on the compressed matrix, i.e., variables in a block should be consecutive in the pivot order.

If the compression information does not introduce fill-in, the quality of the ordering should be similar with and without the activation of the analysis by blocks. If the quality of the ordering is degraded by the analysis by blocks, one should check that compression information is correct. On some types of matrices with many nonzeros per row where the analysis by blocks was observed to sometimes slightly degrade the quality of the ordering when using METIS (ICNTL(7)=5), note that setting the METIS option METIS_NSEPS to 2 (see) generally recovers a good ordering quality (while slightly increasing the time spent in METIS).

Incompatibility: With element entry format ICNTL(5)=1, with Schur complement ICNTL(19) $\neq 0$ and with permutation to a zero-free diagonal and related compressed/constrained ordering for symmetric matrices (ICNTL(6) $\neq 0$, ICNTL(12) $\neq 1$).

ICNTL(16) controls the setting of the number of OpenMP threads, see Subsection 5.22 by MUMPS when the setting of multithreading is not possible outside MUMPS (see Subsection 3.14).

Phase: accessed by the host at the beginning of all phases

Possible values :

0 : (recommended value) nothing is done, MUMPS uses the number of OpenMP threads configured by the calling application.

> 0 : on entry MUMPS sets the number of OpenMP threads to the value of ICNTL(16) and on exit reset the number of OpenMP threads to its value before the MUMPS call.

Other values are treated as 0.

Default value: 0 (nothing is done)

Remarks: Please note that the use of ICNTL(16) should be limited to the case when for some reasons, setting the number of OpenMP threads is not possible outside MUMPS.

ICNTL(17) controls the *MPI to k OMP* reuse factor value, see Subsection 5.23.

Phase: accessed by the host at the beginning of the analysis phase

Possible values :

0 : feature not activated

$k \geq 2$: value of the *MPI to k OMP* reuse factor. The number of MPI processes will be divided by k during computations in the factorization and solve phases while the number of OpenMP threads per MPI process will be multiplied by k

-1 : Automatic setting of *MPI to k OMP* reuse factor

$k < -1$: Try to use up to (but not more than) $|k|$ threads per MPI process (taking into account threads already "attached" to MPI processes)

Default value: 0 (functionality not used)

Remarks:

- This feature has been tested only on Linux platforms.
- To use the *MPI to k OMP* and take the most advantage of it, a way to manage the affinity masks must be provided. We propose two portable solutions, based on a C low-level system API from sched.h, and on the external library HWLOC[14], respectively.

[14] https://www.open-mpi.org/software/hwloc

* To use the C low-level system API (sched.h), please provide the option -DUSE_SCHEDAFFINITY in the OPTF and OPTC variables of the Makefile.inc used to compile MUMPS.
* To use the external library HWLOC, please compile MUMPS with -DUSE_LIBHWLOC (to be also provided in OPTF and OPTC of the Makefile.inc), add -lhwloc (for example in LIBOTHERS in MUMPS Makefile), add the path to the library and to the include directory if needed.

If both -DUSE_LIBHWLOC and -DUSE_SCHEDAFFINITY are used when building MUMPS, HWLOC will be used.

– Let us suppose that the number of MPI processes per node is 18 and that NOMP=2 where NOMP holds the number of OpenMP threads per MPI process (NOMP=omp_get_max_threads()). Examples of values for ICNTL(17):

6 : the effective value of the grouping factor is 6, leading to 12 threads per MPI process and 3 working MPI processes per node

5 : error return INFO(1)=-60 because 5 does not divide 18

−9: the effective value of the grouping factor is 3 (3 is the largest possible value that divides 18 and that is such that $3 * NOMP \leq 9$), leading to 6 threads per MPI process and 6 working MPI processes per node

– For performance analysis and debugging the user may want to check how MPI processes involved in the numerical phases (which depends on PAR) are grouped, the original affinity masks of the MPI processes, and the possibly modified affinity masks of the MPI active workers. The grouping of the MPI processes is printed as a global statistic (unit ICNTL(3) when ICNTL(4) $\geq$ 2). In order to obtain more information on the affinity masks, the preprocessing option -DMTKO_VERBOSE should be provided in the OPTF and OPTC variables of the file Makefile.inc used to compile MUMPS. Additional traces starting with "MPI TO K OMP" will then be printed when ICNTL(2) $>$ 0 and ICNTL(4) $\geq$ 2.

ICNTL(18) defines the strategy for the distributed input matrix (only for assembled matrix, see Subsection 5.4.2).

Phase: accessed by the host during the analysis phase.

Possible values :

0 : the input matrix is centralized on the host (see Subsection 5.4.2.1).

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 5.4.2.2).

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 5.4.2.2).

3 : user directly provides the distributed matrix, pattern and entries, input both for analysis and factorization (see Subsection 5.4.2.2).

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 recommend options 2 or 3. Among them, we recommend 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 5.19).

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 5.19). 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: rank-revealing factorization (ICNTL(56)=1), maximum transversal, scaling, iterative refinement, error analysis and parallel analysis.

Related parameters: ICNTL(7), ICNTL(26)

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 5.19.3), 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, sparse, or distributed) of the right-hand sides

Phase: accessed by the host during the solve phase and before a JOB= 9 call.

Possible variables/arrays involved: RHS, NRHS, LRHS, IRHS_SPARSE, RHS_SPARSE, IRHS_PTR, NZ_RHS, Nloc_RHS, LRHS_loc, IRHS_loc, RHS_loc, INFO(23).

Possible values :

0 : the right-hand side is in dense format in the structure component RHS, NRHS, LRHS (see Subsection 5.18.1)

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.

10, 11 : distributed right-hand side.
The right-hand side is provided distributed in the structure components Nloc_RHS, LRHS_loc, IRHS_loc, RHS_loc (see Subsection 5.18.3).
When provided before a JOB= 9 call, values 10 and 11 indicate which distribution MUMPS should build and return it to the user in IRHS_loc. In this case, the user should provide a workarray IRHS_loc on each MPI process of size at least INFO(23), where INFO(23) is returned after the factorization phase.

- 10 : fill IRHS_loc to minimize internal communications of right-hand side data during the solve phase.

11 : fill IRHS_loc to match the distribution of the solution in case of the distribution of the solution is imposed by MUMPS (ICNTL(21)=1).

In any case, values 10 and 11 have the same meaning entering the solve phase: use the distributed right-hand side in IRHS_loc, RHS_loc without taking into account how the distribution has been computed.

Values different from 0, 1, 2, 3, 10, 11 are treated as 0. For a sparse right-hand side, the recommended value is 1.

Default value: 0 (dense right-hand sides)

Incompatibility: When 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 5.18.1, Subsection 5.18.2 and Subsection 5.18.3. Please note that duplicate entries in the sparse or distributed right-hand sides are summed. A JOB= 9 call can only be done after a successful factorization phase and its results depends on the transpose option ICNTL(9) , which should not be modified between a JOB= 9 and JOB= 3 call. The distributed right-hand side feature enables the user to provide a sparse structured RHS (i.e., a RHS with some empty rows that will be considered equal to zero).

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, SOL_loc, LSOL_loc, INFO(23).

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 5.18.5).

1 : the solution vector is kept distributed on each slave processor in the structure components ISOL_loc and SOL_loc. The distribution of the solution is chosen by MUMPS and is returned in ISOL_loc. The arrays ISOL_loc and SOL_loc must 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 5.18.6).

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 3.16).

Other values are treated as 0 (in-core factorization).

Default value: 0 (in-core factorization)

Related parameters: ICNTL(35) since factors in low-rank form are not written to disk

Remarks: The variables OOC_TMPDIR and OOC_PREFIX are used to indicate the directory and the prefix, respectively, where to store the files containing 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 deallocation phase (JOB= –2 or JOB= –4) is called, except if the Save/Restore feature has been used and the files containing the factors are associated to a saved instance. See Section Subsection 5.21.4).

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 process, see Subsection 5.12 for more details.

Phase: accessed by all processes at the beginning of the factorization phase. If the value is greater than 0 only on the host, then the value on the host is used for all processes, otherwise ICNTL(23) is interpreted locally on each MPI process.

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 process to be allocated

Default value: 0

Related parameters: ICNTL(14), ICNTL(38), ICNTL(39)

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 does not exceed 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 then shared between the main (and often most critical) real/complex internal workarray S holding the factors, the stack of contribution blocks and dynamic workarrays that are used either to expand the S array or to store low-rank dynamic structures.

Lower bounds for ICNTL(23), in case ICNTL(23) is provided only on the host:

- In case of full-rank factors only (ICNTL(35)=0 or 3), a lower bound for 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).
- In case of low-rank factors (ICNTL(35)=1 or 2) only (ICNTL(37)=0), a lower bound for ICNTL(23) (if ICNTL(14), has not been modified since the analysis and ICNTL(38) is a good approximation of the average compression rate of the factors) is given by INFOG(36) if the factorization is in-core (ICNTL(22)=0), and by INFOG(38) if the factorization is out-of-core (ICNTL(22)=1).
- In case of low-rank contribution blocks (CB) only (ICNTL(35)=0,3 and ICNTL(37)=1), a lower bound for ICNTL(23) (if ICNTL(14), has not been modified since the analysis and ICNTL(39) is a good approximation of the average compression rate of the CB) is given by INFOG(44) if the factorization is in-core (ICNTL(22)=0), and by INFOG(46) if the factorization is out-of-core (ICNTL(22)=1).

– In case of low-rank factors and contribution blocks (ICNTL(35)=1,2 and ICNTL(37)=1), a lower bound for ICNTL(23) (if ICNTL(14), has not been modified since the analysis, and ICNTL(38) and ICNTL(39) are good approximations of the average compression rate of respectively the factors and the CB) is given by INFOG(40) if the factorization is in-core (ICNTL(22)=0), and by INFOG(42) if the factorization is out-of-core (ICNTL(22)=1).

Lower bounds for ICNTL(23), in case ICNTL(23) is provided locally to each MPI process:

– Full-rank factors only (ICNTL(35)=0 or 3) ⇒ INFO(15) if the factorization is in-core (ICNTL(22)=0), INFO(17) if the factorization is out-of-core (ICNTL(22)=1).
– Low-rank factors (ICNTL(35)=1 or 2) only (ICNTL(37)=0) ⇒ INFO(30) if the factorization is in-core (ICNTL(22)=0), INFO(31) if the factorization is out-of-core (ICNTL(22)=1).
– Low-rank factors and contribution blocks (ICNTL(35)=1,2 and ICNTL(37)=1) ⇒ is given by INFO(34) if the factorization is in-core (ICNTL(22)=0), INFO(35) if the factorization is out-of-core (ICNTL(22)=1).

The above lower bounds include memory for the real/complex internal workarray S holding the factors and stack of contribution blocks. In case WK_USER is provided, the above quantities should be diminished by the estimated memory for S/WK_USER. This estimated memory can be obtained from INFO(8), INFO(9), or INFO(20) (depending on MUMPS settings) by taking their absolute value, if negative, or by dividing them by 10^6, if positive. See also the paragraph *Recommended values of* LWK_USER below.

If ICNTL(23) is left to its default value 0 then MUMPS will allocate for the factorization phase a workspace based on the estimates computed during the analysis if ICNTL(14) has not been modified since analysis, or larger if ICNTL(14) was increased. Note that even with full-rank factorization, these estimates are only 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 larger than the provided estimations.

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 row will result in error INFO(1)=−10.
1: Null pivot row detection.

Other values are treated as 0.

Default value: 0 (no null pivot row detection)

Related parameters: CNTL(3), CNTL(5), ICNTL(13), ICNTL(25), ICNTL(56)

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

It can be used alone or combined with the rank-revealing option (see ICNTL(56)). In this case CNTL(3) it is used to determine if a pivot should be postponed.

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, PIVNUL_LIST, 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(56), ICNTL(21), ICNTL(24)

Remarks: Null space basis computation can be activated when a zero-pivot detection option (ICNTL(24) $\neq 0$) and/or SVD decomposition on the root node (ICNTL(56) $= 1$) was requested during the factorization and 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 processes (see Subsection 5.18.6). 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 3.19 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 3.19, 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 and 2 then error analysis and iterative refinement are disabled (ICNTL(11) and ICNTL(10))

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

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, $-2\times$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 5.6).

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. The ordering method is set by ICNTL(29) and the ICNTL(7) parameter is meaningless.

Any other values will be treated as 0.

Default value: 1 (sequential computation)

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(28)= 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 5.6.

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.

Other values are treated as 0.

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 5.6.1 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 5.18.4).

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 [55, 51, 18].

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 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 $\mathbf{L}$ factor is discarded.

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 a Schur complement is needed (see ICNTL(19)), or 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, 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 and ICNTL(32)=2, MUMPS currently discards the **L** factors corresponding to full-rank frontal matrices but not of low-rank frontal matrices (except if ICNTL(35)=3). In a future version, discarding all the **L** factors in case of BLR factorization and ICNTL(32)=2 may lead to a further memory reduction.

ICNTL(32) performs the forward elimination of the right-hand sides (Equation (3)) during the factorization (JOB= 2). (see Subsection 5.17).

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), with the computation of entries of the inverse (ICNTL(30)=1), and with BLR factorizations (ICNTL(35)=1,2,3). In such cases, error −43 is raised.

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 5.18.1. 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, the L factor will then always be discarded (even when ICNTL(31)=0) in the case of a full-rank factorization (ICNTL(35)=0) or BLR factorization with full-rank solve (ICNTL(35)=3). In the case of a BLR factorization with ICNTL(35)=1 or 2, only the L factor corresponding to full-rank frontal matrices are discarded in the current version.

We advise to use this option only for a reasonably small 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 3.19).

Although we recommend to compute the determinant on non-singular matrices, null pivot rows (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 row 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) controls the conservation of the OOC files during JOB= –3 (See Subsection 5.21).

Phase: accessed by the host during the Save/Restore files deletion phase (JOB= –3) in case of out-of-core (ICNTL(22)=1).

Possible values :

0: the out-of-core files are marked out for deletion

1: the out-of-core files should not be deleted because another saved instance references them.

Other values are treated as 0.

Default value: 0 (out-of-core files associated to a saved instance are marked out for deletion at the end of the out-of-core file lifetime)

Remarks: MUMPS will delete only the out-of-core files that are referenced in the saved data identified by the value of SAVE_DIR and SAVE_PREFIX. Extra out-of-core files with the same OOC_TMPDIR and OOC_PREFIX are not deleted.

ICNTL(35) controls the activation of the BLR feature (see Subsection 5.20).

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

Possible values :

0 : Standard analysis and factorization (BLR feature is not activated).

1 : BLR feature is activated and automatic choice of BLR option is performed by the software.

2 : BLR feature is activated during both the factorization and solution phases, which allows for memory gains by storing the factors in low-rank.

3 : BLR feature is activated during the factorization phase but not the solution phase, which is still performed in full-rank. As a consequence, the full-rank factors must be kept and no memory gains can be obtained. In an OOC context, (ICNTL(22)=1) this option enables the user to write *all* factors to disk which is not the case with ICNTL(35)=2 since factors in low-rank form are not written to disk.

Other values are treated as 0.

Default value: 0 (standard multifrontal factorization).

Related parameters: CNTL(7) (BLR approximations accuracy), ICNTL(36) (BLR factorization variant), ICNTL(37) (compression of the contribution blocks), ICNTL(38) (estimation of the compression rate of the factors) and ICNTL(39) (estimation of the compression rate of the contribution blocks).

Incompatibility: Note that the activation of the BLR feature is currently incompatible with elemental matrices (ICNTL(5) = 1) (see error -800, subject to change in the future), and when the forward elimination during the factorization is requested (ICNTL(32) = 1), see error -43.

Remarks: If ICNTL(35)=1, then the automatic choice of BLR option is to activate BLR feature during both factorization and solution phases (ICNTL(35)=2). In order to activate the BLR factorization, ICNTL(35) must be equal to 1, 2 or 3 before the analysis, where some preprocessing on the graph of the matrix is needed to prepare the low-rank factorization. The value of ICNTL(35) can then be set to any of the above values on entry to the factorization (e.g., taking into account the values returned by the analysis). On the other hand, if ICNTL(35)=0 at analysis, only ICNTL(35)=0 is allowed for the factorization (full-rank factorization). When activating BLR, it is recommended to set ICNTL(35) to 1 or 2 rather than 3 to benefit from memory gains.

ICNTL(36) controls the choice of BLR factorization variant (see Subsection 5.20).

Phase: accessed by the host during the factorization phase when ICNTL(35)=1, 2 or 3

Possible values :

0 : Standard UFSC variant with low-rank updates accumulation (LUA)

1 : UCFS variant with low-rank updates accumulation (LUA). This variant consists in performing the compression earlier in order to further reduce the number of operations and leads to a better tradeoff between flops reduction and accuracy.

Other values are treated as 0.

Default value: 1 (UCFS variant).

Related parameters: ICNTL(35) and CNTL(1)

Remarks: If numerical pivoting is not required and thus CNTL(1) can be set to 0.0, further performance gains can be expected with the UCFS version.

ICNTL(37) controls the BLR compression of the contribution blocks (see Subsection 5.20).

Phase: accessed by the host during the factorization phase when ICNTL(35)=1, 2 or 3

Possible values :

0 : contribution blocks are not compressed

1 : contribution blocks are compressed, reducing the memory consumption at the cost of some additional operations

Other values are treated as 0.

Default value: 0 (contribution blocks not compressed).

Related parameters: ICNTL(35), CNTL(7), ICNTL(40)

Remarks: This feature should be activated if memory consumption is a primary concern; note that it is likely to increase the factorization time. We recommend to combine this feature with ICNTL(40)=1 since BLR contribution blocks will then also benefit from adaptive precision representation for further storage reductions.

ICNTL(38) estimates compression rate of *LU* factors (see Subsection 5.20).

Phase: accessed by the host during the analysis and the factorization phases when ICNTL(35)=1, 2 or 3

Possible values : between 0 and 1000 (1000 is no compression and 0 is full compression); other values are treated as 0; ICNTL(38)/10 is a percentage representing the typical compression of the factor matrices in BLR fronts: ICNTL(38)/10 $= \frac{\text{compressed factors}}{\text{uncompressed factors}} \times 100$.

Default value: 600 (when factors of BLR fronts are compressed, their size is 60.0% of their full-rank size).

Related parameters: ICNTL(35), CNTL(7)

Remarks: Influences statistics provided in INFO(29), INFO(30), INFO(31), INFOG(36), INFOG(37), INFOG(38), INFOG(39) , but also INFO(32-35) and INFOG(40-43)

ICNTL(39) estimates compression rate of contribution blocks (see Subsection 5.20).

Phase: accessed by the host during the analysis and the factorization phases when ICNTL(35)=1, 2 or 3, and ICNTL(37) = 1

Possible values : between 0 and 1000 (1000 is no compression and 0 is full compression); other values are treated as 0; ICNTL(39)/10 is a percentage representing the typical compression of the CB in BLR fronts: ICNTL(39)/10 $= \frac{\text{compressed CB}}{\text{uncompressed CB}} \times 100$.

Default value: 500 (when CB of BLR fronts are compressed, their size is 50% of their full-rank size).

Related parameters: ICNTL(35), ICNTL(37)

Remarks: Influences statistics provided in INFO(32), INFO(33), INFO(34), INFO(35), INFO(36), INFO(37), INFO(38), INFOG(40), INFOG(41), INFOG(42), INFOG(43) INFOG(44) INFOG(45) INFOG(46) INFOG(47)

ICNTL(40) activates adaptive precision (see Subsection 5.20) Block Low-Rank feature (see Subsection 5.20.4).

Phase: accessed by the host during factorization phase when ICNTL(35)=1, 2 or 3

Possible values :

0 : no adaptive precision BLR

1 : adaptive precision BLR to reduce the memory footprint of the BLR LU factors.

Default value: 0 (no adaptive precision BLR)

Related parameters: ICNTL(35), CNTL(7) and ICNTL(37)

Remarks: In this version, the use of adaptive precision mainly targets storage reductions. The reduced size of the BLR LU factors can, however, also accelerate the triangular solve phase in some cases. With adaptive precision BLR activated on should expect to reduce INFOG(9) since the space needed to *store* the BLR factors can be further reduced.

ICNTL(41-46) reserved in current version

ICNTL(47) controls the possibility to perform a single precision factorization in a double precision MUMPS instance (see Subsection 5.10).

Phase: Accessed by the host during the analysis and factorization phases, only meaningful for double precision MUMPS instances (DMUMPS or ZMUMPS).

Possible values :

0 : nothing done, factorization and solves are performed in the working precision

1 : factorization and solves are performed in single precision

Out-of-range values are treated as 0.

Default value: 0 (factorization is performed in the working precision)

Incompatibility: This feature is not compatible with out-of-core execution (ICNTL(22)), with rank-revealing (ICNTL(56)), with the provision of a user workspace (WK_USER), and with the computation of a Schur complement (ICNTL(19)).

Remarks: The access to ICNTL(47) during the analysis phase only impacts the computed statistics. The value of ICNTL(47) is allowed to change between analysis and factorization, but memory statistics from the analysis will be incorrect. To obtain correct statistics with a new value of ICNTL(47), it is not necessary to perform the whole analysis again, performing a cheap analysis (JOB= 13) is enough.

ICNTL(48) multithreading with tree parallelism (see Subsection 5.26).

Phase: accessed by the host during the analysis phase (need be set after the initialization phase (JOB= –1)) and during the solution phase

Possible values :

0 : not activated

1 : multithreaded tree parallelism activated

Other values are treated as 0.

Default value: 1

Related parameters: if tree parallelism is activated, then the number of threads per MPI process set through the OMP_NUM_THREADS environment variable or ICNTL(16) control parameter should be at least 2. It will influence the choice of the $\mathcal{L}_0$ layer at the analysis phase and should not be modified during the subsequent numerical phases.

Remarks:

- Please note that, once ICNTL(48) $= 1$ is set prior to the analysis phase it will be used for both analysis and factorization phases. Tree parallelism can be switched off (ICNTL(48)=0) prior to the solution phase (JOB= 3).
- The number of threads per MPI process should be identical and if not multithreaded tree parallelism will be switched off during analysis.
- If tree parallelism is not effective during analysis then it cannot be activated during the factorization and solve phases.
- The memory provided with WK_USER will not be used under the $\mathcal{L}_0$ layer making WK_USER not recommended when ICNTL(48) $= 1$.

ICNTL(49) compact workarray id%S at the end of factorization phase (see Subsection 5.24).

Phase: accessed by the host during factorization phase

Possible values :

0 : nothing is done.

1 : compact workarray id%S(MAXS) at the end of the factorization phase while satisfying the memory constraint that might have been provided with ICNTL(23) feature.

2 : compact workarray id%S(MAXS) at the end of the factorization phase. The memory constraint that might have been provided with ICNTL(23) feature does not apply to this process.

Other values are treated as 0.

Default value: 0

Incompatibility: With the use of LWK_USER / WK_USER feature.

Remarks: ICNTL(49)=1,2 might require intermediate memory allocation to reallocate id%S of minimal size. If the memory allocation fails, then a warning is returned and nothing is done. If ICNTL(49)=1 and the memory constraint provided with ICNTL(23) > 0 would not be satisfied then a warning is raised and nothing is done.

ICNTL(50) Stop on user request (see Subsection 5.25).

Phase: accessed by working MPI processes during the factorization and solve phases

Possible values :

0 : nothing is done, MUMPS performs its computations normally.

1 : user wants to cancel the MUMPS execution. When a process detects that ICNTL(50) has been set to 1, it informs the other processes and the computations are cancelled even if the factorization (JOB= 2) or the solve (JOB= 3) are not finished. The error code returned is INFO(1)=−80 if MUMPS was cancelled during the factorization phase, and INFO(1)=−81 if MUMPS was cancelled during the solve phase. INFO(2) holds the rank of the MPI process on which ICNTL(50) was set to 1.

Other values are treated as 0.

Default value: 0

Remarks: In case of non-working host (PAR=0), ICNTL(50) should be set on a working MPI process, i.e. on a process with rank greater than 0 in the communicator COMM passed to MUMPS. Furthermore, if the MUMPS execution was cancelled with ICNTL(50)=1, it is the user's responsibility to reset ICNTL(50) to 0 before a new call to MUMPS.

ICNTL(51) Offload activities on GPUs (see Subsection 5.27).

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

Possible values :

– during analysis:

≤ 0 : nothing done

$k > 0$: modify analysis (increase granularity and modify scheduling of tasks, increase buffer sizes) to expose more work to GPUs.

– during factorization:

≤ 0 : do not use GPU acceleration

$k > 0$: use k GPUs per MPI process

Default value: 0

Remarks:

– $k = 1$ is recommended; k should be smaller or equal to the number of GPUs accessible by each MPI process.

– In case of multiple MPI processes, only the value of ICNTL(51) on the host (MPI process with rank 0) is meaningful and all the MPI processes are assumed to use the same number of GPUs.

– In case of multiple MPI processes per node and $k > 1$, it may happen that a GPU will be shared by several MPI processes.

ICNTL(52-55) reserved in current version

ICNTL(56) detects pseudo-singularities during factorization and factorizes the root node with a rank-revealing method. See Subsection 5.14 for details

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

Possible values :

0 : standard factorization is performed.

1 : Postponing and rank-revealing factorization on root node based on a singular value decomposition.

Values different from 1 are treated as 0.

Default value: 0 (standard factorization)

Related parameters: ICNTL(13), ICNTL(24), ICNTL(25), CNTL(1), CNTL(3)

Remarks: Note that ICNTL(56) must be set before analysis, during which a valid positive value prepares the data for later use of the rank-revealing functionality.

The root is processed sequentially and ICNTL(13) setting is ignored.

Please note that to improve the numerical behaviour of the factorization, the default value of CNTL(1) has been increased in case of activation of the rank-revealing feature. On numerically difficult problems the value of CNTL(1) may be further increased.

CNTL(3) it is used to determine if a pivot should be postponed.

Related data:

mumps_par%INFOG(28) (integer):
INFOG(28) is set during factorization to the deficiency of the matrix. It counts null pivot rows if ICNTL(24) $=1$ and null singular values (ICNTL(56) $=1$).

mumps_par%**PIVNUL_LIST** (integer array, dimension INFOG(28)):
If INFOG(28) $\neq 0$ then PIVNUL_LIST(1:INFOG(28)) will hold, on the host, the row indices corresponding to both the null pivot rows (if ICNTL(24)$=1$, the first entries upto INFOG(51) of PIVNUL_LIST) and the null singular values (ICNTL(56) $=1$).

mumps_par%**SINGULAR_VALUES**(1:mumps_par%**NB_SINGULAR_VALUES**) (real pointer array) which holds, on the host, all the singular values corresponding to SVD decomposition on the root node.

If the matrix was found to be deficient (INFOG(28) >0), the solution phase (JOB= 3) can then be used to either provide a "regular" solution or to compute the null-space basis (see ICNTL(25)).

Incompatibility: With Schur complement feature ICNTL(19)$\neq 0$

ICNTL(57) reserved in current version

ICNTL(58) defines options for symbolic factorization

Phase: accessed by the host during the analysis when centralized ordering is performed, ICNTL(28) = 0, 1

Possible values :

1: Symbolic factorization based on quotient graph, mixing right looking and left looking updates

2: Column count based symbolic factorization based on [36]

Other values are treated as 2.

Default value: 2

Related parameters: ICNTL(7), ICNTL(28)

Remarks: When symbolic factorization is not performed within the ordering (case of ordering given, ICNTL(7)=1 or centralized Metis ordering, ICNTL(7)=5) then symbolic factorization will be automatically performed. When SCOTCH is used, ICNTL(7)=3, a fast block symbolic factorization (exploiting graph separator information) provided within SCOTCH library, libesmumps.a, is used.

ICNTL(59-60) not used in current version

6.2 Real/complex control parameters

mumps_par%**CNTL** is a **real** (also **real** in the complex version) array of dimension 15.

CNTL(1) is the relative threshold for numerical pivoting. See Subsection 3.11

Phase: accessed by the host during the factorization phase.

Possible values :

< 0.0: Automatic choice

$= 0.0$: 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: -1.0 (automatic choice):

0.1: in case of rank-revealing (`ICNTL(56)` $= 1$)

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 :

< 0.0: values < 0 are treated as $\sqrt{\epsilon}$, where ϵ holds the machine precision and depends on the arithmetic version.

≥ 0.0: stopping criterion

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 5.8) 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 2).

CNTL(3) it is used to determine *null pivot rows* when the null pivot row detection option is enabled (`ICNTL(24)` $= 1$) and/or *singularities at the root node* when the rank-revealing option is enabled (`ICNTL(56)` $= 1$). It is also used to determine small pivots whose elimination will be *postponed* if rank-revealing option is enabled (`ICNTL(56)` $= 1$).

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

Possible values : we define the threshold $thres$ as follows

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

$= 0.0$: $thres = \epsilon \times \|\mathbf{A_{pre}}\| \times \sqrt{\mathbf{N_h}}$

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

where A_{pre} is the preprocessed matrix to be factorized (see Equation (5)), N_h number of variables on the deepest branch of the elimination tree, ϵ is the machine precision and $\|.\|$ is the infinite norm.

Default value: 0.0

Related parameters: `ICNTL(24)`, `ICNTL(56)`

Remarks:

- If rank-revealing is enabled (ICNTL(56)=1) then
  - *singular values* smaller than $thres$ are considered as null.
  - $thres$ is also used to automatically define pivot rows that are small (with respect to the infinite norm of its row/column) and that should be *postponed* possibly up to the root node on which rank-revealing will be performed.
  - when *null pivot row* detection is also enabled (ICNTL(24)=1), the threshold value to determine null pivot rows also relies on $thres$.
- When only null pivot row detection is enabled (ICNTL(24)=1). A pivot is considered to be null if the infinite norm of its row/column is smaller than $thres$.

CNTL(4) determines the threshold for static pivoting. See Subsection 3.11

Related parameters: CNTL(1), INFOG(25)

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 pivots whose magnitude is smaller than CNTL(4) will be set to CNTL(4).

$= 0.0$: static pivoting is activated and the threshold value to define a small pivot is determined automatically. In the current version, this threshold is equal to $\sqrt{\epsilon} \times \|\mathbf{A_{pre}}\|$, where $\mathbf{A_{pre}}$ is the preprocessed matrix to be factored (see Equation (5)).

Default value: -1.0 (no static pivoting)

Related parameters: CNTL(1)

Incompatibility: This option is incompatible with null pivot row detection (ICNTL(24) = 1) or with rank-revealing factorization (ICNTL(56) = 1) and will be ignored.

Remarks: By static pivoting (as in [46]) 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). The number of modified pivots is returned in INFOG(25).

CNTL(5) defines the fixation for null pivots and is effective only when null pivot row 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), the pivot column of the $\mathbf{L}$ factors is set to zero and the pivot entry in matrix $\mathbf{D}$ is set to one.
In the unsymmetric case (SYM = 0), the fixation is automatically set to a large positive value and the pivot row of the $\mathbf{U}$ factors is set to zero.

> 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(6) is not used in the current version.

CNTL(7) defines the precision of the dropping parameter used during BLR compression (see Subsection 5.20).

Phase: accessed by the host during the factorization phase when ICNTL(35)=1, 2 or 3

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: The value of CNTL(7) 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 [23] _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 scale the matrix or or let the solver automatically preprocess (e.g., scale) the input matrix.

Note that, depending on the application, gains can be expected even with small values (close to machine precision) of CNTL(7).

CNTL(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))		
		Centralized		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))	Seq. or parallel	Sequential only	Sequential or parallel
Schur complement	(ICNTL(19))	All options except parallel analysis and Rank-Revealing, ICNTL(56) $\neq$ 0		All options except parallel analysis and ICNTL(56) $\neq$ 0
Block Low-Rank	(ICNTL(35)=1,2,3)	All options except ICNTL(32)=1 (fwd in facto)	Not available	All options except ICNTL(32)=1 (fwd in facto)

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 basis	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.

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. In case the BLR feature is activated (ICNTL(35)=1, 2 or 3), RINFO(3) represents the theoretical number of operations for the standard full-rank factorization. Note that RINFO(3) is updated dynamically during the factorization process, and thus can be monitored by the user to have an idea of the number of operations done, see Subsection 5.25.

RINFO(4) - after factorization: The effective number of floating-point operations on the processor for the elimination process. It is equal to RINFO(3) when the BLR feature is not activated (ICNTL(35)=0) and will typically be smaller than RINFO(3) when the BLR feature is activated and leads to compression.

RINFO(5) - after analysis: if the user decides to perform an out-of-core factorization (ICNTL(22)=1), then a rough estimation of the size of the disk space in MegaBytes of the files written by the concerned processor is provided in RINFO(5). If the analysis is full-rank (ICNTL(35)=0 for the analysis step), then the factorization is necessarily full-rank so that RINFO(5) is computed for a full-rank factorization (ICNTL(35)=0 also for the factorization). If ICNTL(35)=1, 2 or 3 at analysis, then RINFO(5) is computed assuming a low-rank (in-core) storage of the factors of the BLR fronts during the factorization (ICNTL(35)=1 or 2 during factorization). In case ICNTL(35)=1, 2 or 3 at analysis and the factors are stored in full-rank format (ICNTL(35)=0 or 3 for the factorization), we refer the user to INFO(3) in order to obtain a rough estimate of the necessary disk space for the concerned processor.

The effective size in MegaBytes of the files written by the current processor will be returned in RINFO(6), but only after the factorization. The total estimated disk space (sum of the values of RINFO(5) over all processors) is returned in RINFOG(15).

RINFO(6) - after factorization: in the case of an out-of-core execution (ICNTL(22)=1), the size in MegaBytes of the disk space used by the files written by the concerned processor is provided. The total disk space (for all processors) is returned in RINFOG(16).

RINFO(7) - after each job: The size (in MegaBytes) of the file used to save the data on the processor (See Subsection 5.21).

RINFO(8) - after each job: The size (in MegaBytes) of the MUMPS structure.

RINFO(9) (*experimental, subject to change in the future*) - after factorization: number of floating-point operations offloaded to the accelerator(s) by the local MPI process. See also RINFOG(22).

RINFO(10) (*experimental, subject to change in the future*) - after factorization: time spent in operations offloaded to the accelerator(s), including communication, on the local MPI process. See also RINFOG(23).

RINFO(11) - RINFO(40) are not used in the current version.

mumps_par%**INFO** is an integer array of dimension 80. 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. In particular, after successfully saving or restoring an instance (call to MUMPS with JOB= 7 or JOB= 8), INFO(1) will be 0 even if INFO(1) was different from 0 at the moment of saving the MUMPS instance to disk.

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, assuming the factors are stored in full-rank format (ICNTL(35)=0 or 3 during factorization). If INFO(3) is negative, then its absolute value corresponds to *millions* of real/complex entries used to store the factor matrices. Assuming that the factors will be stored in full-rank format during the factorization (ICNTL(35)=0 or 3), 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. See also RINFO(5).

Note that, when all factors are discarded (ICNTL(31)=1), INFO(3) corresponds to the factors storage if factors were not discarded (rather than 0). However, if only the **L** factor is discarded (case of forward substitution during factorization, ICNTL(32)=1, or case of ICNTL(31)=2), then INFO(3) corresponds to the factor storage excluding **L**.

The effective size of the real/complex space needed to store the factors will be returned in INFO(9) (see below), but only after the factorization. Furthermore, after an out-of-core factorization (ICNTL(22)=1), the size of the disk space for the files written by the local processor is returned in RINFO(6). Finally, the total estimated size of the full-rank factors for all processors (sum of the INFO(3) values over all processors) is returned in INFOG(3).

INFO(4) - after analysis: Estimated integer space needed on the processor for factors (assuming a full-rank storage for the factors)

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

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

INFO(7) - after analysis: Minimum estimated size of the main internal integer workarray IS to run the numerical factorization *in-core*.

INFO(8) - after analysis: Minimum estimated size of the main internal real/complex workarray S to run the numerical factorization *in-core* when factors are stored full-rank (ICNTL(35)=0 or 3). If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray. It is also the estimated minimum size of LWK_USER in that case, if the user provides WK_USER.

INFO(9) - after factorization: Size of the real/complex space used on the processor to store the factor matrices, possibly including low-rank factor matrices (ICNTL(35)=1 or 2). If negative, then the absolute value corresponds to *millions* of real/complex entries used to store the factor matrices. Finally, the total size of the factor matrices for all processors (sum of the INFO(9) values over all processors) is returned in INFOG(9).

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.) If rank-revealing option is on (ICNTL(56)=1), the inertia might be incorrect if INFO(1) $\geq$ 16. Furthermore, when ICNTL(24) is set to 1 and SYM=1 or 2, or when ICNTL(56)=1, INFOG(12) excludes the null[15] pivots, even if their sign

[15] i.e., whose magnitude is smaller than the tolerance defined by CNTL(3).

is negative. In other words, a pivot cannot be both null and negative. 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. For real symmetric matrices, see also INFO(40) and INFOG(50), which provide the local (resp. global) number of negative pivots among the null pivots detected when ICNTL(24) (and/or ICNTL(56)) is activated.

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 perform full-rank numerical phases (factorization/solve) *in-core* (ICNTL(22)=0 for the factorization). The maximum and sum over all processors are returned respectively in INFOG(16) and INFOG(17). See also INFO(22) which provides the actual memory that was needed but only after factorization.

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. The maximum and sum over all processors are returned respectively in INFOG(18) and INFOG(19).

INFO(17) - after analysis: estimated size in MegaBytes (millions of bytes) of all working space to run the numerical phases *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). See also INFO(22) which provides the actual memory that was needed but only after factorization.

INFO(18) - after factorization: local number of null pivot rows detected locally when ICNTL(24)=1 or ICNTL(56)= 1.

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

INFO(20) - after analysis: Estimated size of the main internal real/complex workarray S to run the numerical factorization *out-of-core*. If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray. It is also the estimated minimum size of LWK_USER in that case, if the user provides WK_USER.

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 part of the memory effectively used from the workarray WK_USER, in the case where WK_USER is provided. The maximum and sum over all processors are returned respectively in INFOG(21) and INFOG(22). The difference between estimated and effective memory may results from numerical pivoting difficulties, parallelism and BLR effective compression rates.

INFO(23) - after factorization: total number of pivots eliminated on the processor. It may be used in the case of a distributed right-hand side (see ICNTL(20)) and/or of a distributed solution (see ICNTL(21)).

INFO(24) - after analysis: estimated number of entries in the factor matrices 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). The total number of entries in the factor matrices for all processors (sum of the INFO(24) values over all processors) is returned in INFOG(20).

INFO(25) - after factorization: number of tiny pivots (number of pivots modified by static pivoting) detected on the processor (see INFOG(25) for the the total number of tiny pivots).

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 in INFOG(30) and INFOG(31), respectively).

INFO(27) - after factorization: effective number of entries in factor matrices assuming full-rank factorization has been performed. If negative, then the absolute value corresponds to *millions* of entries in the factors. Note that in case full-rank storage of factors (ICNTL(35)=0 or 3), we have INFO(27)=INFO(9) in the unsymmetric case and INFO(27) $\leq$ INFO(9) in the symmetric case. The sum of INFO(27) over all processors is available in INFOG(29).

INFO(28) - after factorization: effective number of entries in factors on the processor taking into account BLR compression. If negative, then the absolute value corresponds to *millions* of entries in the factors. It is equal to INFO(27) when BLR functionality (see ICNTL(35)) is not activated or leads to no compression.

INFO(29) - after analysis: minimum estimated size of the main internal real/complex workarray S to run the numerical factorization *in-core* when factors are stored low-rank (ICNTL(35)=1,2). If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray. It is also the estimated minimum size of LWK_USER in that case, if the user provides WK_USER.

INFO(30) and INFO(31) - after analysis: estimated size in MegaBytes (millions of bytes) of all working space to perform low-rank numerical phases (factorization/solve) with low-rank factors (ICNTL(35)=1,2) and estimated compression rate given by ICNTL(38).

- —– (30) *in-core* factorization and solve The maximum and sum over all processors are returned respectively in INFOG(36) and INFOG(37).
- —– (31) *out-of-core* factorization and solve The maximum and sum over all processors are returned respectively in INFOG(38) and INFOG(39).

See also INFO(22) which provides the actual memory that was needed but only after factorization. Numerical pivoting difficulties and the effective compression of the factors (in case ICNTL(35)=1,2) typically impact the difference between estimated and effective memory.

INFO(32) - after analysis: minimum estimated size of the main internal real/complex workarray S to run the numerical factorization *in-core* when factors and contribution blocks are stored low-rank (ICNTL(35)=1,2 and ICNTL(37)=1). If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray. It is also the estimated minimum size of LWK_USER in that case, if the user provides WK_USER.

INFO(33) - after analysis: minimum estimated size of the main internal real/complex workarray S to run the numerical factorization *out-of-core* when factors and contribution blocks are stored low-rank (ICNTL(35)=1,2 and ICNTL(37)=1). If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray. It is also the estimated minimum size of LWK_USER in that case, if the user provides WK_USER.

INFO(34) and INFO(35) - after analysis: estimated size in MegaBytes (millions of bytes) of all working space to perform low-rank numerical phases (factorization/solve) with low-rank factors and low-rank contribution blocks (ICNTL(35)=1,2 and ICNTL(37)=1) and estimated compression rates given by ICNTL(38) and ICNTL(39) relatively.

- —– (34) *in-core* factorization and solve. The maximum and sum over all processors are returned respectively in INFOG(40) and INFOG(41).
- —– (35) *out-of-core* factorization and solve The maximum and sum over all processors are returned respectively in INFOG(42) and INFOG(43).

See also INFO(22) which provides the actual memory that was needed but only after factorization.

INFO(36) - after analysis: minimum estimated size of the main internal real/complex workarray S to run the numerical factorization *out-of-core* when contribution blocks are stored low-rank (ICNTL(35)=0,3 and ICNTL(37)=1). If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray. It is also the estimated minimum size of LWK_USER in that case, if the user provides WK_USER.

INFO(37) and INFO(38) - after analysis: estimated size in MegaBytes (millions of bytes) of all working space to perform low-rank numerical phases (factorization/solve) with low-rank contribution blocks only (ICNTL(35)=0,3 and ICNTL(37)=1) and estimated compression rate given by ICNTL(39).

- —– (37) *in-core* factorization and solve. The maximum and sum over all processors are returned respectively in INFOG(44) and INFOG(45).
- —– (38) *out-of-core* factorization and solve The maximum and sum over all processors are returned respectively in INFOG(46) and INFOG(47).

See also INFO(22) which provides the actual memory that was needed but only after factorization.

INFO(39) - after factorization: effective size of the main internal real/complex workarray S (allocated internally or by the user when WK_USER is provided) to run the numerical factorization. If negative, then the absolute value corresponds to *millions* of real/complex entries needed in this workarray.

INFO(40) - after factorization: can only be nonzero for real symmetric matrices, in case the null pivot row detection (see ICNTL(24)) feature or rank-revealing (see ICNTL(56)) is activated. INFO(40) is the number of negative pivots among the null pivots/deficiency detected. Note that, for singular matrices, INFO(40) may vary from one run to another due to floating-point rounding effects. A pivot counted in INFO(12), the number of negative non-null pivots, will not be counted in INFO(40). See also INFOG(28) which provides the number of null pivots/deficiency over all processors and INFOG(50), which provides the number of negative null pivots over all processors.

INFO(41) - INFO(80) 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. In case the BLR feature is activated (ICNTL(35)=1, 2 or 3), RINFOG(3) represents the theoretical number of operations for the standard full-rank factorization.

RINFOG(4) to RINFOG(8) - after solve with error analysis: Only returned if ICNTL(11) $= 1$ or 2. See description in Subsection 5.9 .

RINFOG(9) to RINFOG(11) - after solve with error analysis: Only returned if ICNTL(11) $= 1$. See description in Subsection 5.9 .

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 effective number of floating-point operations (on all processors) for the elimination process. It is equal to RINFOG(3) when the BLR feature is not activated (ICNTL(35)=0) and will typically be smaller than RINFOG(3) when the BLR functionality is activated and leads to compression.

RINFOG(15) - after analysis: if the user decides to perform an out-of-core factorization (ICNTL(22)=1), then a rough estimation of the total size of the disk space in MegaBytes of the files written by all processors is provided in RINFOG(15). If the analysis is full-rank (ICNTL(35)=0 for the analysis step), then the factorization is necessarily full-rank so that RINFOG(15) is computed for a full-rank factorization (ICNTL(35)=0 also for the factorization). If ICNTL(35)=1, 2 or 3 at analysis, then RINFOG(15) is computed assuming a low-rank (in-core) storage of the factors of the BLR fronts during the factorization (ICNTL(35)=2 during

factorization). In case ICNTL(35)=1, 2 or 3 for the analysis and the factors will be stored in full-rank format (ICNTL(35)=0 or 3 for the factorization), we refer the user to INFOG(3) in order to obtain a rough estimate of the necessary disk space for all processors.

The effective size in Megabytes of the files written by all processors will be returned in RINFOG(16), but only after the factorization.

RINFOG(16) - after factorization: in the case of an out-of-core execution (ICNTL(22)=1), the total size in MegaBytes of the disk space used by the files written by all processors is provided.

RINFOG(17) - after each job: sum over all processors of the sizes (in MegaBytes) of the files used to save the instance (See Subsection 5.21).

RINFOG(18) - after each job: sum over all processors of the sizes (in MegaBytes) of the MUMPS structures.

RINFOG(19) - after factorization: smallest pivot in absolute value selected during factorization of the preprocessed matrix A_{pre} (see Equation (5)) and considering *ALSO* small pivots selected as null-pivots (see ICNTL(24)) and pivots on which static pivoting (see CNTL(4)) is effective.

RINFOG(20) - after factorization: smallest pivot in absolute value selected during factorization of the preprocessed matrix A_{pre} (see Equation (5)) and *NOT* considering small pivots selected as null-pivots (see ICNTL(24)) and pivots on which static pivoting (see CNTL(4)) is effective. A huge value of RINFOG(20) indicates that all pivots were either null-pivots or pivots on which static pivoting was performed.

RINFOG(21) (*experimental, subject to change in the future*) - after factorization: largest pivot in absolute value selected during factorization of the preprocessed matrix A_{pre} (see Equation (5)).

RINFOG(22) - after factorization: total number of floating-point operations offloaded to the accelerator(s) by all MPI processes. See also RINFO(9).

RINFOG(23) - after factorization: average (over all MPI processes) time spent in operations offloaded to the accelerator(s), including communication. See also RINFO(10).

RINFOG(24) - RINFOG(40) are not used in the current version.

mumps_par%**INFOG** is an integer array of dimension 80. 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. In particular, after successfully saving or restoring an instance (call to MUMPS with JOB= 7 or JOB= 8), INFOG(1) will be 0 even if INFOG(1) was different from 0 at the moment of saving the MUMPS instance to disk.

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 factors, assuming the factors are stored in full-rank format (ICNTL(35)=0 or 3). If INFOG(3) is negative, then its absolute value corresponds to *millions* of real/complex entries used to store the factor matrices. Assuming that the factors will be stored in full-rank format during the factorization (ICNTL(35)=0 or 3), a rough estimation of the size of the disk space in bytes of the files written all processors can be obtained by multiplying INFOG(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. See also RINFOG(15).

Note that, when all factors are discarded (ICNTL(31)=1), INFOG(3) corresponds to the factors storage if factors were not discarded (rather than 0). However, if only the **L** factor is discarded (case of forward substitution during factorization, ICNTL(32)=1, or case of ICNTL(31)=2), then INFO(3) corresponds to the factor storage excluding **L**.

The effective size of the real/complex space needed will be returned in INFOG(9) (see below), but only after the factorization. Furthermore, after an out-of-core factorization, the size of the disk space for the files written by all processors is returned in RINFOG(16).

INFOG(4) - after analysis: total (sum over all processors) estimated integer workspace to store the factor matrices (assuming a full-rank storage of the factors). If INFOG(4) is negative, then its absolute value corresponds to *millions* of integer entries used 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 or if analysis by block was performed (ICNTL(15))).

INFOG(9) - after factorization: total (sum over all processors) real/complex workspace to *store* the factor matrices, possibly including low-rank factor matrices (ICNTL(35)=2). The unit to count the size of the workspace is the number of real/complex and thus INFOG(9) is independent of the value of ICNTL(47). It is however sensitive to the value of ICNTL(40) because adpative precision may reduce the space needed 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 factor matrices. With adaptive precision BLR activated (ICNTL(40)=1) on should expect to reduce INFOG(9) since the space needed to store the BLR factors can be further reduced.

INFOG(10) - after factorization: total (sum over all processors) integer workspace to store the factor matrices. If negative the absolute value corresponds to *millions* of integer entries in the 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.) If rank-revealing option is on (ICNTL(56)=1), the inertia might be incorrect if INFO(1) $\geq$ 16. Furthermore, when ICNTL(24) is set to 1 and SYM=1 or 2, or when ICNTL(56)=1, INFOG(12) excludes the null[16] 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. For real symmetric matrices, see also INFOG(40), provides the number of negative pivots among the null pivots detected when ICNTL(24) (and/or ICNTL(56)) is activated.

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 full-rank factorization *in-core* for a given value of ICNTL(14).

- —— (16) : max over all processors

[16] i.e., whose magnitude is smaller than the tolerance defined by CNTL(3).

- —— (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) : max over all processors
- —— (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 assuming full-rank factorization. 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) : max over all processors
- —— (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 full-rank factorization *out-of-core* (ICNTL(22) $\neq$ 0) for a given value of ICNTL(14).

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

INFOG(28) - after factorization: size of the null space, resulting from detection of null pivot rows (see ICNTL(24) and CNTL(3)) and of singularities on the root node (see ICNTL(56) and CNTL(3)).

INFOG(29) - after factorization: effective number of entries in the factor matrices (sum over all processors) assuming that full-rank factorization has been performed. If negative, then the absolute value corresponds to *millions* of entries in the factors. Note that in case the factor matrices are stored full-rank (ICNTL(35)=0 or 3), we have INFOG(29)=INFOG(9) in the unsymmetric case and INFOG(29) $\leq$ INFOG(9) in the symmetric case.

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) - after factorization: effective number of entries in the factors (sum over all processors) taking into account BLR factor compression. If negative, then the absolute value corresponds to millions of entries in the factors. It is equal to INFOG(29) when BLR functionality (see ICNTL(35)) is not activated or leads to no compression.

INFOG(36), INFOG(37), INFOG(38), and INFOG(39) - after analysis: estimated size (in million of bytes) of all MUMPS internal data for running low-rank factorization with low-rank factors for a given value of ICNTL(14) and ICNTL(38).

- *in-core*
  - —— (36) : max over all processors
  - —— (37) : sum over all processors.
- *out-of-core*
  - —— (38) : max over all processors
  - —— (39) : sum over all processors.

INFOG(40), INFOG(41), INFOG(42), and INFOG(43) - after analysis: estimated size (in million of bytes) of all MUMPS internal data for running low-rank factorization with low-rank factors and low-rank contribution blocks for a given value of ICNTL(14), ICNTL(38) and ICNTL(39).

- *in-core*
  - —— (40) : value on the most memory consuming processor.
  - —— (41) : sum over all processors.
- *out-of-core*
  - —— (42) : value on the most memory consuming processor.
  - —— (43) : sum over all processors.

INFOG(44), INFOG(45), INFOG(46), and INFOG(47) - after analysis: estimated size (in million of bytes) of all MUMPS internal data for running low-rank factorization with low-rank low-rank contribution blocks only for a given value of ICNTL(14) and ICNTL(39).

- *in-core*
  - —— (44) : value on the most memory consuming processor.
  - —— (45) : sum over all processors.
- *out-of-core*
  - —— (46) : value on the most memory consuming processor.
  - —— (47) : sum over all processors.

INFOG(48) - INFOG(49) are reserved.

INFOG(50) - after factorization: can only be nonzero for symmetric matrices, in case the null pivot row detection (see ICNTL(24)) feature or rank-revealing (see ICNTL(56)) is activated. INFOG(50) is the total, over all MPI processes, number of negative pivots among the null pivots/deficiency detected. Note that, for singular matrices, INFOG(50) may vary from one run to another due to floating-point rounding effects. A pivot counted in INFOG(12), the number of negative non-null pivots, will not be counted in INFOG(50). See also INFOG(28), the number of null pivots/deficiency, and INFO(40), the local static for INFOG(50) on a given MPI process.

INFOG(51) - after factorization: size of the null space detected with null pivot row detection (ICNTL(24)=1).

INFOG(52) - INFOG(80) are not used in the current version.

8 Error and warning 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/NZ, NNZ_loc/NZ_loc or $\sum$ **NNZ_loc**/$\sum$ **NZ_loc** are out of range. INFO(2)=NNZ/NZ, NNZ_loc/NZ_loc or $\sum$ **NNZ_loc**/$\sum$ **NZ_loc**.

-3 MUMPS was called with an invalid value for JOB. This may happen if the analysis (JOB= 1) was not performed (or failed) before the factorization (JOB= 2), or the factorization was not performed (or failed) 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 for other cases of error or incompatibilities. 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 is 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 LWK_USER is provided LWK_USER should be increased.

-10 Numerically singular matrix, or zero pivot encountered. INFO(2) holds the number of eliminated pivots. This error may occur if the matrix is numerically singular, or if the matrix is non-singular but numerical pivoting is necessary to factorize the matrix and has been switched off (SYM=1 or CNTL(1)=0).

-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.

-18 The blocking size for multiple RHS (ICNTL(27)) is too large and may lead to an integer overflow. This error may only occurs for very large matrices with large values of ICNTL(27) (e.g., several thousands). INFO(2) provides an estimate of the maximum value of ICNTL(27) that should be used.

-19 The maximum allowed size of working memory ICNTL(23) is too small to run the factorization phase and should be increased. If INFO(2) is positive, then the size in terms of the number of entries (real/complex) that are missing 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.

-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, JCN_loc or A_loc
17	IRHS_loc
18	RHS_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 too small. This error may occur in case a distributed solution has been requested. When ICNTL(21)=1, LSOL_loc should be greater or equal to 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, $1 \leq i \leq$ 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 of the Schur feature 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. If ICNTL(25) is incompatible with ICNTL(xx), 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 ≤ 0. INFO(2) contains the value of NRHS.

-46 NZ_RHS ≤ 0. This is currently not allowed with 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.

-54 The analysis phase (JOB= 1) was called with ICNTL(35)=0 but the factorization phase was called with ICNTL(35)=1, 2 or 3. In order to perform the factorization with BLR compression, please perform the analysis phase again using ICNTL(35)=1, 2 or 3 (see the documentation of ICNTL(35)).

-55 During a call to MUMPS including the solve phase with distributed right-hand side, either LRHS_loc was detected to be smaller than Nloc_RHS and INFO(2)=LRHS_loc, or Nloc_RHS was not equal to zero on the non working host (PAR=0) and INFO(2)=–Nloc_RHS.

-56 During a call to MUMPS including the solve phase with distributed right-hand side and distributed solution, RHS_loc and SOL_loc point to the same workarray but LRHS_loc $<$ LSOL_loc. INFO(2)=LRHS_loc.

-57 During a call to MUMPS analysis phase with a block format (ICNTL(15) $\neq$ 0), an error in the interface provided by the user was detected. INFO(2) holds additional information about the issue:

INFO(2)	issue
1	NBLK is incorrect (or not compatible with BLKPTR size), or -ICNTL(15) is not compatible with N
2	BLKPTR is not provided or its content is incorrect
3	BLKVAR if provided should be of size N

-58 During a call to MUMPS with ICNTL(48)=1, an error occurred. INFO(2) holds additional information about the issue:

- INFO(2)=0: ICNTL(48) was equal to 1 at analysis, but compilation is without OpenMP enabled. You should recompile MUMPS with OpenMP enabled.
- INFO(2)=$k, k > 0$: ICNTL(48) is active but the number of threads available for the current phase (factorization or solve) is different from k, the number of threads available at analysis. Please call the current phase with the same number of threads as for the analysis.
- INFO(2)=$-100 - k, k > 0$: ICNTL(48) is active but the number of threads effectively created during the main parallel region of the factorization that exploits multithreaded tree parallelism is different from the one obtained with omp_get_max_threads(), and used during analysis to prepare the work. k is the number of threads effectively obtained in the parallel region, retrieved with omp_get_num_threads(). Please check your OpenMP environment (OMP_DYNAMIC, OMP_THREAD_LIMIT, ...) to see why k is different from the value returned by omp_get_num_threads().

-60 Error in the use of the *MPI to k OMP* feature. INFO(2) holds additional information on the type of error detected:

- INFO(2)=-1: the value of ICNTL(17) is not compatible with the number of MPI processes per node.
- INFO(2)=-2: is not compiled with OpenMP while ICNTL(17) has been set to a value greater than 1.
- INFO(2)=-3: the number of OpenMP threads is not the same on every MPI process.

-61 Error related to the *MPI to k OMP* feature: error during an affinity mask get/set operation.

-62 Error related to the *MPI to k OMP* feature: allocation issue.

-63 Error related to the *configuration file* feature (see Section 11): the file specified by the environment variable MUMPS_CONFIG_FILE cannot be opened. If INFO(2)>0, the reason is that the length of the filename provided in the environment variable MUMPS_CONFIG_FILE is too large compared to INFO(2), the maximum length authorized by the package. If INFO(2)=-1, the reason is that the open statement failed (e.g. file does not exist, insufficient permissions, file name with quotes or double quotes not authorized ...).

-64 Error related to the *configuration file* feature (see Section 11): a read operation did not succeed (I/O error, ...). INFO(2) contains the line number where the error occurred.

-65 Error related to the *configuration file* feature (see Section 11): a syntax error is present in the line to be read (e.g., parenthesis error, missing equal sign, unknown parameter, ...). INFO(2) contains the line number where the error occurred.

-66 Error related to the *configuration file* feature (see Section 11): a file specified at line INFO(2) could not be opened or read correctly.

-67 Error related to the *configuration file* feature (see Section 11): an allocation could not be performed. INFO(2) contains the line number where the error occurred. This error can occur, for example, if a line in the configuration file provides a file for BLKVAR (through the keyword BLKVAR_FILE) and BLKVAR could not be allocated or is already allocated on entry to MUMPS.

-69 The size of the default Fortran INTEGER datatype does not match the size of MUMPS_INT. INFO(2) indicates the size of MUMPS_INT, which depends on the -DINTSIZE64 during the build process. If INFO(2)=4, the Fortran sources were compiled with an option (e.g. -i8, or -fdefault-integer-8) to make all Fortran integers be 64-bit, but -DINTSIZE64 was missing from the OPTCMakefile.inc variable. If INFO(2)=8, then Fortran sources were compiled in a standard way (32-bit integers) but but -DINTSIZE64 was used. Please fix the Fortran default integer size and/or the use of the -DINTSIZE64 and reinstall and recompile everything. In particular, make sure that you are not using an old version of the file mumps_int_def.h.

-70 During a call to MUMPS with JOB= 7, the file specified to save the current instance, as derived from SAVE_DIR and/or SAVE_PREFIX, already exists. Before saving an instance into this file, it should be first suppressed (see JOB= –3). Otherwise, a different file should be specified by changing the values of SAVE_DIR and/or SAVE_PREFIX.

-71 An error has occurred during the creation of one of the files needed to save MUMPS data (JOB= 7).

-72 Error while saving data (JOB= 7); a write operation did not succeed (e.g., disk full, I/O error, ...). INFO(2) is the size that should have been written during that operation.

If INFO(2) is negative, then its absolute value should be multiplied by 1 million.

-73 During a call to MUMPS with JOB= 8, one parameter of the current instance is not compatible with the corresponding one in the saved instance.

INFO(2) points to the incorrect parameter in the table below:

INFO(2)	parameter
1	*fortran* version (after/before 2003)
2	integer size(32/64 bit)
3	saved instance not compatible over MPI processes
4	number of MPI processes
5	arithmetic
6	SYM
7	PAR

-74 The file resulting from the setting of SAVE_DIR and SAVE_PREFIX could not be opened for restoring data (JOB= 8). INFO(2) is the rank of the process (in the communicator COMM) on which the error was detected.

-75 Error while restoring data (JOB= 8); a read operation did not succeed (e.g., end of file reached, I/O error, ...). INFO(2) is the size still to be read. 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.

-76 Error while deleting the files (JOB= –3); some files to be erased were not found or could not be suppressed. INFO(2) is the rank of the process (in the communicator COMM) on which the error was detected.

-77 Problem with SAVE_DIR and/or SAVE_PREFIX: if INFO(2)=0 then the problem is that neither SAVE_DIR nor the environment variable MUMPS_SAVE_DIR are defined. If INFO(2) $>$ 0, the environment variable MUMPS_SAVE_DIR is defined but its length is larger than the maximum authorized length indicated by INFO(2). If INFO(2) $<$ 0, the environment variable MUMPS_SAVE_PREFIX is defined but its length is larger that the maximum authorized length indicated by -INFO(2).

-78 Problem of workspace allocation during the restore step. The size still to be allocated 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.

-79 MUMPS could not find a Fortran file unit to perform I/O's. INFO(2) provides additional information on the error:

- INFO(2)=1: the problem occurs in the analysis phase, when attempting to find a free Fortran unit for the WRITE_PROBLEM feature (see Subsection 5.4.3).
- INFO(2)=2: the problem occurs during a call to MUMPS with JOB= 7, 8 or -3 (Save-Restore feature, see Subsection 3.21).
- INFO(2)=3: the problem is related to the *configuration file* feature (see Section 11): no free file unit could be found to be associated to the configuration file specified by the environment variable MUMPS_CONFIG_FILE.
- INFO(2)=4: the problem is related to the *configuration file* feature (see Section 11): no new unit could be found to open a file specified by a line in the configuration file.

-80 The factorization phase was cancelled per user request, see ICNTL(50) and Subsection 5.25). INFO(2) is the MPI rank (or one of the MPI ranks) on which the user asked to cancel the execution.

-81 The solution phase was cancelled per user request, see ICNTL(50) and Subsection 5.25). INFO(2) is the MPI rank (or one of the MPI ranks) on which the user asked to cancel the execution.

-88 An error occurred during SCOTCH ordering. INFO(2) holds the error number returned by SCOTCH.

-89 An error occurred during SCOTCH kway-partitioning in SCOTCHFGRAPHPART. The error code returned by SCOTCH is provided in INFO(2). We suggest making the METIS package available to MUMPS.

-90 Error in out-of-core management. See the error message returned on output unit ICNTL(1) for more information.

-92 Incompatible values of ICNTL(47) and ICNTL(xx):

- xx=19: the ICNTL(47) feature is currently incompatible with Schur (ICNTL(19)). Please swith off one of the two features.
- xx=22: the ICNTL(47) feature is currently incompatible with out-of-core execution (ICNTL(22)). Please switch off one of the two features.
- xx=56: the ICNTL(47) feature is currently incompatible with rank revealing (ICNTL(56)). Please switch off one of the two features.
- xx=-1: the ICNTL(47) feature is currently incompatible with user workspace provision LWK_USER / WK_USER.

-100 The number of GPU devices is too small with respect to the number of GPU devices requested through ICNTL(51). INFO(2) is the number of GPU devices available.

-102 The initialization of the cuBLASXT handle (function cublasXtCreate) failed. INFO(2) contains the error code returned by cublasXtCreate.

-104 The call to cublasXtSetBlockDim failed. INFO(2) holds the error code returned by cublasXtSetBlockDim.

-105 The initialization of the XKBlas library failed. INFO(2) holds the error code returned by xkblas_init.

-106 The value of omp_get_max_threads as returned by Fortran and C routines is different. This might be due to an inconsistent setting of OpenMP in OPTF and OPTC when compiling MUMPS.

-107 An error occurred in the rank-revealing feature. INFO(2) holds either the value out of range of ICNTL(56) or the error code returned by the LAPACK routines.

-800 Temporary error associated to the current MUMPS release, subject to change or disappearance in the future. If INFO(2)=5, then this error is due to the fact that the elemental matrix format (ICNTL(5)=1) is currently incompatible with a BLR factorization (ICNTL(35)$\neq$0). If INFO(2)=17, then this error is due to the fact that the *MPI to k OMP* feature (see ICNTL(17)) is currently not compatible with GPU usage (ICNTL(51)).

A positive value of INFO(1) is associated with a successful MUMPS execution but with a warning. The corresponding warning message 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.

+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 ICNTL(49)=1,2 and not enough memory to compact id%S at the end of the factorization. If this is due to memory limitation given by user (ICNTL(23) $>$ 0) then it is advised to reset ICNTL(23) to zero or to set ICNTL(49)=2.

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

+16 Warning return from rank-revealing feature (ICNTL(56)). The values of the inertia (INFOG(12)) and/or the determinant (ICNTL(33)) might not be consistent with the number of singularities. In the context of rank-revealing the inertia and the determinant are computed with RR (rank-revealing) LU. The deficiency found by RR LU, returned in INFO(2), is different from that computed with rank-revealing feature (ICNTL(56)). Furthermore, if INFO(2) $<$ INFOG(28), then the last entries of PIVNUL_LIST could not be computed (they are then set to -1).

+ Combinations of the above warnings will correspond to summing the constituent warnings. For example, if an MPI process exits the package with INFO(1)=6, this indicates that both warnings +2 and +4 occurred on this MPI process. In case several warnings occur on an MPI process, INFO(2) corresponds to the warning that occurred last. Finally, in case of multiple MPI processes, INFOG(1) combines the warnings raised on the different MPI processes (for example, INFOG(1) will be equal to 5 if INFO(1)=1, 1 and 4 on MPI processes with ranks 0, 1 and 2, respectively). In that case, INFOG(2) is simply set to the number of MPI processes on which a warning occurred and the values of INFO(1) and INFO(2) on each MPI process can be checked for more detailed information.

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 5 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:

```
typedef struct
   {
    int sym, par, job;
    int comm_fortran; /* Fortran communicator */
    int icntl[60];
    real cntl[15];
    int n;
    /* Assembled entry */
    int nz; int64_t nnz; int *irn; int *jcn; real/complex *a;
    /* Distributed entry */
    int nz_loc; int *irn_loc; int *jcn_loc; real/complex *a_loc;
    /* Element entry */
    int nelt; int *eltptr; int *eltvar; real/complex *a_elt;
    /* Ordering, if given by user, Metis options */
    int *perm_in;
    int metis_options[40];
    /* Scaling */
    real/complex *colsca; real/complex *rowsca;
    /* RHS, solution, output data and statistics */
    real/complex *rhs, *redrhs, *rhs_sparse, *sol_loc, *rhs_loc;
    int *irhs_sparse, *irhs_ptr, *isol_loc, *irhs_loc;
    int nrhs, lrhs, lredrhs, nz_rhs, lrhs_loc, nloc_rhs, lsol_loc, nsol_loc;
    int info[80],infog[80];
    real rinfo[40], rinfog[40];
    int *sym_perm, *uns_perm;
    /* mapping, null pivots */
    int *mapping, *pivnul_list;
    /* 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[32];
    char ooc_tmpdir[1024], ooc_prefix[256];
    char write_problem[1024];
    char save_dir[1024], save_prefix[256];
    /* Internal parameters */
    int instance_number;
   } [SDCZ]MUMPS_STRUC_C;
```

Figure 5: 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.

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

An example of calling MUMPS from C for a complex assembled problem is given in Subsection 12.3. 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 5.4.2.3), 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.

Remark that Fortran compilers often provide an option to make all default INTEGER datatypes 64-bit integers. In that case, one should add the option -DINTSIZE64 to the Makefile.inc variable OPTC during the installation of MUMPS to indicate that the default Fortran INTEGER should match a 64-bit integer of type int64_t. An error -69 will be raised if Fortran INTEGER is 64-bit (resp. 32-bit) and -DINTSIZE64 was (resp. was not) used. See also the content of the file mumps_int_def.h generated during the build process.

When including MUMPS headers files from a C application, one can then check at compilation time the preprocessing constants MUMPS_INTSIZE32 and MUMPS_INTSIZE64 to know how MUMPS_INT was defined. At runtime, one can simply check the value of sizeof(MUMPS_INT).

Since not all C compilers define the complex datatype (this 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 single 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 3.13.

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 single-MPI 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 5.11.

Please refer to the report [33] 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. Some parameters related to distributed environments should not be modifed. For example, the solution should always be centralized and id.ICNTL(21) should thus remain to its default value, 0. 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 Accessing control parameters from a configuration file

This feature gives access to MUMPS main control parameters from a configuration file. The parameters which can be accessed are :

- the **standard** real and integer control parameters: CNTL and ICNTL,
- the **internal** real and integer (standard or 64-bit) control parameters (DKEEP, KEEP and KEEP8).

METIS_OPTIONS and WRITE_PROBLEM can also be set through this configuration file.

Furthermore, the analysis by blocks can be activated and BLKPTR and BLKVAR can be read from files.

To activate this *configuration file* feature, the environment variable MUMPS_CONFIG_FILE should be set to the name of the configuration file, without quotes, including its complete path (maximum length: 1023 characters). If the MUMPS_CONFIG_FILE environment variable is not set, the configuration file is ignored. The configuration file is then read by the host processor each time MUMPS is called. This means that its content may be modified by the main application each time MUMPS is called. The configuration

file is read only by the host processor and thus only one configuration file is needed. and the parameters present in the configuration file will be set to the same value on each processor. This gives the possibility to the user to change settings between MUMPS calls. The consistency between the content of configuration files at each step is thus the responsibility of the user.

In case an error occurs with the configuration file (I/O issue, file not found, syntax error, ...), MUMPS will return to the calling application with a negative error code in INFO(1), see the error codes -63, -64, -65, -66, -67 and -79 in Section 8.

Example and syntax of the configuration file

We provide below a small example of a configuration file to illustrate the simplicity and flexibility of its syntax.

```
# Example of configuration file
# Force out-of-core factorization
  ICNTL(22)=1
# Block low-rank feature
#   activate BLR
  ICNTL(35)=1
  ICNTL(36)=1
#   define low-rank threshold
  CNTL(7)=1E-7
# Write input test matrix to file
  WRITE_PROBLEM=path_and_file_name
# Change an entry of METIS_OPTIONS
  METIS_OPTIONS(16)=3
# ... which corresponds to NSEPS=3 in metis 5.1
  BLKPTR_FILE=/tmp/blkptr.txt
  BLKVAR_FILE=/tmp/blkvar.txt
  ICNTL(15)=1
# ... to use the analysis by blocks and read
# ... BLKTPR and BLKVAR from files
```

Comments on the syntax of the configuration file:

- # is for comments. The # character must appear as the first character of a comment line.
- The simple syntax VARIABLE = VALUE must be used to set parameters.
- White spaces are ignored.
- Uppercase or lowercase letters are allowed.
- Further description of specific parameters:
  - WRITE_PROBLEM: the file name should be provided without quotes and should include the complete path. The file name should not include special characters '(',')','#','='. See also Subsection 5.4.3.
  - METIS_OPTIONS(see ICNTL(7)=5): it allows the user to set a Metis option to a value different from its default for the call to Metis_NodeND done by MUMPS in case Metis is chosen during the ordering phase (ICNTL(7)=5).
  - BLKPTR_FILE=filename, BLKVAR_FILE=filename: the file names should be provided without quotes and should include the complete path. The file names should not include the special characters '(',')','#','='. The files corresponding to BLKPTR and BLKVAR should have the exact same format as the ones in Subsection 5.4.3:
    - for BLKPTR: the number of blocks NBLK followed by BLKPTR(1:NBLK+1), one integer per line, in text form
    - for BLKVAR: N different integers between 1 and N, with one integer per line, in text form

    Note that, when building BLKPTR, NBLK is also set.

Configuration file and third party software

The configuration file can be used when calling MUMPS from a third party software, such as, for example, PETSc, in order to set MUMPS parameters that are not present in the interface of the third party software. A particular attention should be paid by the user to preserve the consistency between the parameters set in the configuration file and the data allocated by the third party software, in order to avoid errors or unexpected behaviour. For example, the configuration file should not set ICNTL(18)=1 (asking for a Schur complement) without the third party software having allocated memory to return a Schur complement, nor control parameters for matrix (e.g. ICNTL(5)), RHS or solution formats (e.g., ICNTL(5), ICNTL(20), ICNTL(21)) that are incompatible with data provided by the third party software.

If the PETSc routines concerning MUMPS (*MatMumpsSetCntl* and *MatMumpsSetIcntl*) are used simultaneously with the configuration file, the configuration file settings prevail. This is ensured by the fact that the configuration file is read at each call to MUMPS.

12 Examples of use of MUMPS

12.1 An assembled problem

An example program illustrating a possible use of MUMPS on assembled DOUBLE PRECISION problems is given Figure 6.

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

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

we could have as input

```
5                : N
12               : 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
```

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

12.2 An elemental problem

An example of a driver to use MUMPS for element DOUBLE PRECISION problems is given in Figure 7.

```
      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 6: Example program using MUMPS on an assembled DOUBLE PRECISION problem

```
      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 7: Example program using MUMPS on an elemental DOUBLE PRECISION problem.

The calling sequence is similar to that for the assembled problem in Subsection 12.1 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

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

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.

12.3 An example of calling MUMPS from C

An example of a driver to use MUMPS from C is given in Figure 8.

```
/* 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 8: Example program using MUMPS from C on an assembled problem.

12.4 An example of calling MUMPS from fortran using the Save/Restore feature and Out Of Core

An example program illustrating a possible use of the Save/Restore feature combined with Out Of Core:

```
      PROGRAM MUMPS_TEST_SAVE_RESTORE
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INCLUDE 'mumps_struc.h'
      TYPE (CMUMPS_STRUC) mumps_par_save, mumps_par_restore
      INTEGER IERR, I
      CALL MPI_INIT(IERR)
C Define a communicator for the package.
      mumps_par_save%COMM = MPI_COMM_WORLD
C  Initialize an instance of the package
C  for L U factorization (sym = 0, with working host)
      mumps_par_save%JOB = -1
      mumps_par_save%SYM = 0
      mumps_par_save%PAR = 1
      CALL CMUMPS(mumps_par_save)
      IF (mumps_par_save%INFOG(1).LT.0) THEN
       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
     &        "  mumps_par_save%INFOG(1)= ", mumps_par_save%INFOG(1),
     &        "  mumps_par_save%INFOG(2)= ", mumps_par_save%INFOG(2)
       GOTO 500
      END IF
C  Define problem on the host (processor 0)
      IF ( mumps_par_save%MYID .eq. 0 ) THEN
        READ(5,*) mumps_par_save%N
        READ(5,*) mumps_par_save%NZ
        ALLOCATE( mumps_par_save%IRN ( mumps_par_save%NZ ) )
        ALLOCATE( mumps_par_save%JCN ( mumps_par_save%NZ ) )
        ALLOCATE( mumps_par_save%A( mumps_par_save%NZ ) )
        DO I = 1, mumps_par_save%NZ
          READ(5,*) mumps_par_save%IRN(I),mumps_par_save%JCN(I)
     &          ,mumps_par_save%A(I)
        END DO
      END IF
C  Activate OOC
      mumps_par_save%ICNTL(22)=1
C  Call package for factorization
      mumps_par_save%JOB = 4
      CALL CMUMPS(mumps_par_save)
      IF (mumps_par_save%INFOG(1).LT.0) THEN
       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
     &        "  mumps_par_save%INFOG(1)= ", mumps_par_save%INFOG(1),
     &        "  mumps_par_save%INFOG(2)= ", mumps_par_save%INFOG(2)
       GOTO 500
      END IF
C  Call package for save
      mumps_par_save%JOB = 7
      mumps_par_save%SAVE_DIR="/tmp"
      mumps_par_save%SAVE_PREFIX="mumps_simpletest_save"
      CALL CMUMPS(mumps_par_save)
      IF (mumps_par_save%INFOG(1).LT.0) THEN
       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
```

```
 52      &        "  mumps_par_save%INFOG(1)= ", mumps_par_save%INFOG(1),
 53      &        "  mumps_par_save%INFOG(2)= ", mumps_par_save%INFOG(2)
 54        GOTO 500
 55       END IF
 56 C  Deallocate user data
 57       IF ( mumps_par_save%MYID .eq. 0 )THEN
 58         DEALLOCATE( mumps_par_save%IRN )
 59         DEALLOCATE( mumps_par_save%JCN )
 60         DEALLOCATE( mumps_par_save%A   )
 61       END IF
 62 C  Destroy the instance (deallocate internal data structures)
 63       mumps_par_save%JOB = -2
 64       CALL CMUMPS(mumps_par_save)
 65 C  Now mumps_par_save has be destroyed
 66 C  We use a new instance mumps_par_restore to finish the computation
 67
 68 C  Define a communicator for the package on the new instance.
 69       mumps_par_restore%COMM = MPI_COMM_WORLD
 70 C  Initialize a new instance of the package
 71 C  for L U factorization (sym = 0, with working host)
 72       mumps_par_restore%JOB = -1
 73       mumps_par_restore%SYM = 0
 74       mumps_par_restore%PAR = 1
 75       CALL CMUMPS(mumps_par_restore)
 76       IF (mumps_par_restore%INFOG(1).LT.0) THEN
 77        WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
 78      &        "  mumps_par_restore%INFOG(1)= ",
 79      &        mumps_par_restore%INFOG(1),
 80      &        "  mumps_par_restore%INFOG(2)= ",
 81      &        mumps_par_restore%INFOG(2)
 82        GOTO 500
 83       END IF
 84 C  Call package for restore with OOC feature
 85       mumps_par_restore%JOB = 8
 86       mumps_par_restore%SAVE_DIR="/tmp"
 87       mumps_par_restore%SAVE_PREFIX="mumps_simpletest_save"
 88       CALL CMUMPS(mumps_par_restore)
 89       IF (mumps_par_restore%INFOG(1).LT.0) THEN
 90        WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
 91      &        "  mumps_par_restore%INFOG(1)= ",
 92      &        mumps_par_restore%INFOG(1),
 93      &        "  mumps_par_restore%INFOG(2)= ",
 94      &        mumps_par_restore%INFOG(2)
 95        GOTO 500
 96       END IF
 97 C  Define rhs on the host (processor 0)
 98       IF ( mumps_par_restore%MYID .eq. 0 ) THEN
 99         ALLOCATE( mumps_par_restore%RHS ( mumps_par_restore%N  ) )
100         DO I = 1, mumps_par_restore%N
101            READ(5,*) mumps_par_restore%RHS(I)
102         END DO
103       END IF
104 C  Call package for solution
105       mumps_par_restore%JOB = 3
106       CALL CMUMPS(mumps_par_restore)
```

```
      IF (mumps_par_restore%INFOG(1).LT.0) THEN
       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
     &            "  mumps_par_restore%INFOG(1)= ",
     &        mumps_par_restore%INFOG(1),
     &            "  mumps_par_restore%INFOG(2)= ",
     &        mumps_par_restore%INFOG(2)
        GOTO 500
      END IF
C  Solution has been assembled on the host
      IF ( mumps_par_restore%MYID .eq. 0 ) THEN
        WRITE( 6, * ) ' Solution is ',
     &        (mumps_par_restore%RHS(I),I=1,mumps_par_restore%N)
      END IF
C  Deallocate user data
      IF ( mumps_par_restore%MYID .eq. 0 )THEN
        DEALLOCATE( mumps_par_restore%RHS )
      END IF
C  Delete the saved files
C  Note mumps_par_restore%ICNTL(34) is kept to default (0) to suppress
C  also the OOC files.
      mumps_par_restore%JOB = -3
      CALL CMUMPS(mumps_par_restore)
C  Destroy the instance (deallocate internal data structures)
      mumps_par_restore%JOB = -2
      CALL CMUMPS(mumps_par_restore)
 500  CALL MPI_FINALIZE(IERR)
      STOP
      END
```

The MUMPS instance mumps_par_save 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 the factorization is done using Out Of Core (ICNTL(22) = 1) on all processors to MUMPS with JOB= 4. The instance mumps_par_save is saved by calling MUMPS with JOB= 7, a call to MUMPS with JOB= –2 is performed to deallocate the data structures used by the instance mumps_par_save.

The MUMPS instance mumps_par_restore is initialized by calling MUMPS with JOB= –1. The instance mumps_par_restore is restore at the same state as mumps_par_save was by calling MUMPS with JOB= 8. The rest of the problem is read in by the host (in the component RHS), and the solution is computed in RHS with a call on all processors to MUMPS with JOB= 3. Finally, a call to MUMPS with JOB= –3 is performed to deallocate the data structures used by the instance mumps_par_restore and all files used for restarting (OOC and Save/Restore) are suppressed because ICNTL(34) = 0.

12.5 An example of calling MUMPS from C using the Save/Restore feature

An example of a driver to use MUMPS from C :

```
 1 /* Example program using the C interface to the
 2  * double real arithmetic version of MUMPS, dmumps_c.
 3  * We solve the system A x = RHS with
 4  *   A = diag(1 2) and RHS = [1 4]^T
 5  * Solution is [1 2]^T */
 6 #include <stdio.h>
 7 #include <string.h>
 8 #include "mpi.h"
 9 #include "dmumps_c.h"
10 #define JOB_INIT -1
11 #define JOB_END -2
12 #define USE_COMM_WORLD -987654
13
14 #if defined(MAIN_COMP)
15 /*
16  * Some Fortran compilers (COMPAQ fort) define "main" in
17  * their runtime library while a Fortran program translates
18  * to MAIN_ or MAIN__ which is then called from "main".
19  * We defined argc/argv arbitrarily in that case.
20  */
21 int MAIN__();
22 int MAIN_()
23   {
24     return MAIN__();
25   }
26
27 int MAIN__()
28 {
29   int argc=1;
30   char * name = "c_example_save_restore";
31   char ** argv ;
32 #else
33 int main(int argc, char ** argv)
34 {
35 #endif
36   DMUMPS_STRUC_C id_save,id_restore;
37   MUMPS_INT n = 2;
38   MUMPS_INT8 nnz = 2;
39   MUMPS_INT irn[] = {1,2};
40   MUMPS_INT jcn[] = {1,2};
41   double a[2];
42   double rhs[2];
43
44   int error = 0;
45 /* When compiling with -DINTSIZE64, MUMPS_INT is 64-bit but MPI
46    ilp64 versions may still require standard int for C interface. */
47 /* MUMPS_INT myid, ierr; */
48   int myid, ierr;
49 #if defined(MAIN_COMP)
50   argv = &name;
51 #endif
52   ierr = MPI_Init(&argc, &argv);
53   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 MUMPS save instance. Use MPI_COMM_WORLD */
  id_save.comm_fortran=USE_COMM_WORLD;
  id_save.par=1; id_save.sym=0;
  id_save.job=JOB_INIT;
  dmumps_c(&id_save);
  /* Define the problem on the host */
  if (myid == 0) {
    id_save.n = n; id_save.nnz =nnz; id_save.irn=irn; id_save.jcn=jcn;
    id_save.a = a;
  }
#define ICNTL(I) icntl[(I)-1] /* macro s.t. indices match documentation */
  /* No outputs */
  id_save.ICNTL(1)=-1; id_save.ICNTL(2)=-1;
  id_save.ICNTL(3)=-1; id_save.ICNTL(4)=0;
  /* Call the MUMPS package on the save instance (analyse and factorization). */
  id_save.job=4;
  dmumps_c(&id_save);

  /* MUMPS save feature on the save instance. */
  strcpy(id_save.save_prefix,"csave_restore");
  strcpy(id_save.save_dir,"/tmp");
  if (myid == 0) {
    printf("Saving MUMPS instance in %s with prefix %s.\n",
        id_save.save_dir, id_save.save_prefix);
  }
  id_save.job=7;
  dmumps_c(&id_save);
  if (id_save.infog[0]<0) {
    printf("\n (PROC %d) ERROR RETURN: \tINFOG(1)= %d\n\t\t\t\tINFOG(2)= %d\n",
        myid, id_save.infog[0], id_save.infog[1]);
    error = 1;
  } else if (myid == 0) {
    printf("                                                    DONE\n\n");
  }

  /* Terminate the save instance. */
  id_save.job=JOB_END;
  dmumps_c(&id_save);



  if (!error) {
    /* Initialize MUMPS restore instance. Use MPI_COMM_WORLD */
    id_restore.comm_fortran=USE_COMM_WORLD;
    id_restore.par=1; id_restore.sym=0;
    id_restore.job=JOB_INIT;
    dmumps_c(&id_restore);
    /* Define the rhs on the host */
    if (myid == 0) {
      id_restore.rhs = rhs;
    }
```

```
109
110     /* No outputs */
111     id_save.ICNTL(1)=-1; id_save.ICNTL(2)=-1;
112     id_save.ICNTL(3)=-1; id_save.ICNTL(4)=0;
113
114     /* MUMPS restore feature on restore instance. */
115     if (myid == 0) {
116       printf("Restoring MUMPS instance in %s with prefix %s.\n",
117           id_save.save_dir, id_save.save_prefix);
118     }
119     strcpy(id_restore.save_prefix,"csave_restore");
120     strcpy(id_restore.save_dir,"/tmp");
121     id_restore.job=8;
122     dmumps_c(&id_restore);
123     if (id_save.infog[0]<0) {
124       printf("\n (PROC %d) ERROR RETURN: \tINFOG(1)= %d\n\t\t\t\tINFOG(2)= %d\n",
125           myid, id_save.infog[0], id_save.infog[1]);
126       error = 1;
127     } else if (myid == 0) {
128       printf("                                                   DONE\n\n");
129     }
130   }
131
132   if (!error) {
133     /* Call the MUMPS package on restore instance (solve). */
134     if (myid == 0) {
135       printf("Calling MUMPS package (solve).\n");
136     }
137     id_restore.job=3;
138     dmumps_c(&id_restore);
139     if (id_save.infog[0]<0) {
140       printf("=> (PROC %d) ERROR RETURN: \tINFOG(1)= %d\n\t\t\t\tINFOG(2)= %d\n",
141           myid, id_save.infog[0], id_save.infog[1]);
142       error = 1;
143     } else if (myid == 0) {
144       printf("                                                   DONE\n\n");
145     }
146
147     /* Deletes the saved and the OOC files. */
148     if (myid == 0) {
149       printf("Removing save files.\n");
150     }
151     id_restore.job=-3;
152     dmumps_c(&id_restore);
153     if (id_save.infog[0]<0) {
154       printf("=> (PROC %d) ERROR RETURN: \tINFOG(1)= %d\n\t\t\t\tINFOG(2)= %d\n",
155           myid, id_save.infog[0], id_save.infog[1]);
156       error = 1;
157     } else if (myid == 0) {
158       printf("                                                   DONE\n\n");
159     }
160
161     /* Terminate the restore instance. */
162     id_restore.job=JOB_END;
163     dmumps_c(&id_restore);
```

```
164   }
165
166   if (myid == 0) {
167     if (!error) {
168       printf("Solution is : (%8.2f  %8.2f)\n", rhs[0],rhs[1]);
169     } else {
170       printf("An error has occured, please check error code returned by MUMPS.\n");
171     }
172   }
173   ierr = MPI_Finalize();
174   return 0;
175 }
```

13 License

Copyright 1991-2026 CERFACS, CNRS, ENS Lyon, INP Toulouse, Inria,
Mumps Technologies, University of Bordeaux.

This version of MUMPS is provided to you free of charge. It is
released under the CeCILL-C license
(see doc/CeCILL-C_V1-en.txt, doc/CeCILL-C_V1-fr.txt, and
https://cecill.info/licences/Licence_CeCILL-C_V1-en.html),
except for variants of AMD ordering and [sdcz]MUMPS_TRUNCATED_RRQR
derived from the LAPACK package distributed under BSD 3-clause
license (see headers of ana_orderings.F and [sdcz]lr_core.F),
and except for the external and optional ordering PORD provided
in a separate directory PORD (see PORD/README for License information).

You can acknowledge (using references [1] and [2]) the contribution
of this package in any scientific publication dependent upon the use
of the package. Please use reasonable endeavours to notify the authors
of the package of this publication.

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

[2] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent and T. Mary,
Performance and scalability of the block low-rank multifrontal
factorization on multicore architectures,
ACM Transactions on Mathematical Software,
Vol 45, Issue 1, pp 2:1-2:26 (2019)

As a counterpart to the access to the source code and rights to copy,
modify and redistribute granted by the license, users are provided only
with a limited warranty and the software's author, the holder of the
economic rights, and the successive licensors have only limited
liability.

In this respect, the user's attention is drawn to the risks associated
with loading, using, modifying and/or developing or reproducing the
software by the user in light of its specific status of free software,
that may mean that it is complicated to manipulate, and that also
therefore means that it is reserved for developers and experienced
professionals having in-depth computer knowledge. Users are therefore
encouraged to load and test the software's suitability as regards their
requirements in conditions enabling the security of their systems
and/or data to be ensured and, more generally, to use and operate it
in the same conditions as regards security.

The fact that you are presently reading this means that you have had
knowledge of the CeCILL-C license and that you accept its terms.

14 Credits

This version of MUMPS has been developed by employees of

CERFACS, CNRS, EDF, ENS Lyon, Toulouse INP-IRIT, Inria,
Mumps Technologies, Sorbonne University and University of Bordeaux:
Emmanuel Agullo, Patrick Amestoy, Maurice Bremond, Alfredo Buttari, Philippe
Combes, Marie Durand, Aurelia Fevre, Matthieu Gerest, Abdou Guermouche,
Antoine Jego, Guillaume Joslin, Jacko Koster, Jean-Yves L'Excellent, Theo Mary,
Pierre-Etienne Polet, Stephane Pralet, Chiara Puglisi, Francois-Henry Rouet,
Wissam Sid-Lakhdar, Tzvetomila Slavova, Bora Ucar and Clement Weisbecker.

Since January 2019, the MUMPS solver is maintained by Mumps Technologies
(http://mumps-tech.com).

We are grateful to Caroline Bousquet, Indranil Chowdhury, Christophe Daniel,
Iain Duff, Vincent Espirat, Gilles Moreau, Gregoire Richard, Alexis Salzman,
Miroslav Tuma and Christophe Voemel who have been contributing to this project.

We are also grateful to Juergen Schulze for letting us distribute PORD developed
at the University of Paderborn. We thank Eddy Caron for the administration of a
server that has been used during many years for MUMPS.

We want to thank the French ANR programme, the European community,
Airbus Group-IW, Altair, ANSYS, CINES, EDF, EMGS, ESI Group, FFT/Hexagon,
Keysight, LBNL, LSTC, Michelin, NEC, SAFRAN, SAMTECH, Shell, Siemens, Synopsys,
Total and University of Southern California for their support.
We also thank LBNL, LSTC, PARALLAB and the Rutherford Appleton Laboratory
for research discussions that have certainly influenced this work.

Finally we want to thank the institutions that have provided access to their
parallel machines: Centre Informatique National de l'Enseignement Superieur
(CINES), CERFACS, CALMIP ("Centre Interuniversitaire de Calcul" located
in Toulouse), Centre Blaise Pascal de Simulation et de Modélisation Numérique,
Institut du Developpement et des Ressources en Informatique Scientifique (IDRIS),
Lawrence Berkeley National Laboratory, Laboratoire de l'Informatique du
Parallelisme, Inria, and PARALLAB.

References

[1] E. Agullo. *On the Out-of-core Factorization of Large Sparse Matrices*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.

[2] P. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jezequel, J.-Y. L'Excellent, and T. Mary. Mixed precision low-rank approximations and their application to block low-rank LU factorization. *IMA Journal of Numerical Analysis*, 43:2198–2227, July 2023.

[3] P. Amestoy, A. Buttari, N. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé. Five-Precision GMRES-based iterative refinement. *SIAM Journal on Matrix Analysis and Applications*, pages 529–552, 45(1) 2024.

[4] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé. Combining sparse approximate factorizations with mixed precision iterative refinement. *ACM Transactions on Mathematical Software*, 49(1):1–29, 2023.

[5] P. Amestoy, A. Jego, J.-Y. L'Excellent, T. Mary, and G. Pichon. BLAS-based Block Memory Accessor with Applications to Mixed Precision Sparse Direct Solvers. Preprint submitted to publicationin ACM Transactions on Mathematical Software, Apr. 2025.

[6] P. Amestoy, A. Jego, J.-Y. L'Excellent, T. Mary, and G. Pichon. Efficient mixed-precision memory-bound BLAS based on memory accessors with applications to sparse direct solvers. In *Minisymposium Double, Single or Half the Fun - Mixed and Reduced Precision Computing in SIAM CSE Projects*, Fort Worth, United States of America, March 3-7 2025.

[7] 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.

[8] 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.

[9] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. On the complexity of the Block Low-Rank multifrontal factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740, 2017.

[10] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. *ACM Transactions on Mathematical Software*, 45:2:1–2:26, 2019.

[11] 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.

[12] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputer Applications*, 3:41–59, 1989.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, United States, May 2013.

[35] M. Gerest. *Using Block Low-Rank compression in mixed precision for sparse direct linear solvers*. Ph.D. thesis, Sorbonne University, Nov. 2023.

[36] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 15:1075–1091, 1994.

[37] 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.

[38] A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.

[39] 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.

[40] N. J. Higham and T. Mary. Solving Block Low-Rank Linear Systems by LU Factorization is Numerically Stable. *IMA Journal of Numerical Analysis*, 42(2):951–980, 2021.

[41] 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.

[42] P. Knight and D. Ruiz. A fast algorithm for matrix balancing. *IMA Journal of Numerical Analysis*, 33(3):1029–1047, octobre 2012.

[43] P. A. Knight, D. Ruiz, and B. Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM Journal on Matrix Analysis and Applications*, 35(3):931–955, 2014.

[44] J.-Y. L'Excellent. *Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects*. Habilitation à diriger des recherches, École normale supérieure de Lyon, Sept. 2012.

[45] 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.

[46] 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.

[47] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[48] T. Mary. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability*. PhD thesis, Université de Toulouse, November 2017.

[49] F. Pellegrini. SCOTCH and LIBSCOTCH 5.0 User's guide. Technical Report, LaBRI, Université Bordeaux I, 2007.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

[56] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.

[57] C. Weisbecker. *Improving multifrontal solvers by means of algebraic block low-rank representations*. PhD thesis, Institut National Polytechnique de Toulouse, Oct. 2013.