

Université Paul Sabatier Toulouse III

Maîtrise Ingénierie Mathématique



Training Report

Complex version and validation of **MUMPS**

(**M**ultifrontal **M**assively **P**arallel **S**olver)

by

Caroline Bousquet - Christophe Daniel

WN/PA/02/34

Acknowledgements

We particularly would like to thank Iain Duff, leader of Parallel Algorithms Team in CERFACS, to have accepted us as trainee in his team.

We thank Luc Giraud, researcher at CERFACS, to have proposed us this training course.

We specially thank Patrick Amestoy, teacher and researcher at ENSEEIHT, to have guided us in this project and for his advices.

We thank Christof Voemel and Jean-Christophe Rioual for their help.

We also would like to thank all members of the team for their kindness and their reception.

Contents

Introduction	1
1 Generation of different versions of MUMPS	3
1.1 Hierarchy of generation	3
1.2 Used syntax	4
1.3 Generation of the double precision real version from the <i>SRC</i> file	5
1.4 Generation of the simple precision real version	5
1.5 Generation of the simple precision complex version	6
1.6 Generation of the double precision complex version	7
2 Validation	7
2.1 Coverage	7
2.1.1 Interest of coverage	7
2.1.2 Presentation of FCAT	8
2.2 Tester	12
2.2.1 Tester interest	12
2.2.2 General structure of the tester	12
2.2.3 Numbering of tests and matrix	16
2.2.4 Structure of a single test	18
2.2.5 Using tester	18
2.2.6 Results obtained with the tester	20
2.2.7 Results of coverage	22
3 Complex validation	22
Conclusion	25
A Scripts for generation of different versions of MUMPS	27
A.1 Double precision real version	27
A.2 Single precision real version	27
A.3 Single precision complex version	29
A.4 Double precision complex version	31
B Numbering of tests	33
C Analysis coverage	41
C.1 Example of a script using FCAT for a larger code	41
C.2 Result of the coverage	41

Introduction

For our project, we must have familiarized ourselves with the software : MUMPS¹ [2, 8, 7, 9].

MUMPS (“Multifrontal Massively Parallel Solver”) is a package for solving linear systems of equations $\mathbf{Ax} = \mathbf{b}$, where the matrix \mathbf{A} is sparse and can be either unsymmetric, symmetric positive definite, or general symmetric. MUMPS uses a multifrontal technique which is a direct method based on either the LU or the LDL^T factorization of the matrix. MUMPS exploits both parallelism arising from sparsity in the matrix \mathbf{A} and from dense factorizations kernels.

The main features of the MUMPS package include the solution of the transposed system, input of the matrix in assembled format (distributed or centralized) or elemental format, error analysis, iterative refinement, scaling of the original matrix, estimate of rank deficiency and null space basis, return of Schur complement, and the possibility for the user to input a given ordering. Several instances of MUMPS can be handled simultaneously.

The software is written in Fortran 90. It requires MPI [10, 5] for message passing and makes use of BLAS, BLACS [4, 6], and ScaLAPACK subroutines [3].

MUMPS distributes the work tasks among the processors, but an identified (host) processor is required to perform the analysis phase, distribute the incoming matrix to the other (slave) processors in the case where the matrix is centralized, collect the solution, and generally oversee the computation.

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

1. **Analysis.** The host performs an approximate minimum degree algorithm[1] based on the symmetrized pattern $\mathbf{A} + \mathbf{A}^T$, and carries out symbolic factorization. A mapping of the multifrontal computational graph is then computed, and symbolic information is transferred from the host to the other processors. Using this information, the processors estimate the memory necessary for factorization and solution.
2. **Factorization.** The original matrix is first distributed to processors that will participate in the numerical factorization. The numerical factorization on each frontal matrix is conducted by a master processor (determined by the analysis phase) and one or more slave processors (determined dynamically). Each processor allocates an array for contribution blocks and factors; the factors must be kept for the solution phase.

¹available at the web address <http://www.enseeiht.fr/apo/MUMPS>

3. **Solution.** The right-hand side \mathbf{b} is broadcast from the host to the other processors. These processors compute the solution \mathbf{x} using the (distributed) factors computed during Step 2, and the solution is assembled on the host.

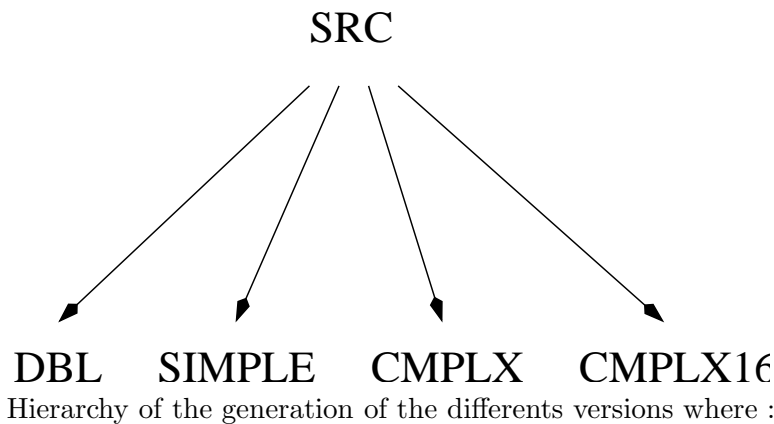
Our job was, on the one hand to generate a complex version of MUMPS, on the other hand to develop an automatic validation of the code. All experiments were done on the SGI Origin 2000 located at CERFACS.

1 Generation of different versions of MUMPS

The purpose of this job is to generate single precision real and complex (single and double precision) versions of MUMPS from only one unique set of source codes written for double precision real.

1.1 Hierarchy of generation

In the *SRC* directory, there is all the source codes are written for double precision real number. From this directory, we can generate the four versions of MUMPS with preprocessing.



- DBL means DOUBLE PRECISION
- SIMPLE means REAL
- CMPLX means COMPLEX
- CMPLX16 means COMPLEX DOUBLE PRECISION.

For example, we would the version in simple precision real, we have to :

- Create the *SIMPLE* directory;
- Create the Makefile in this directory :

```
ARCH = SGI
PREC = SIMPLE
# Precision could be DBL, SIMPLE, CMPLX, CMPLX16
all:init
init:Makefile.ma41 Makefile.inc
    make -f Makefile.ma41
```

```

Makefile.ma41:../SRC/Makefile.ma41
    cp -f ../SRC/Makefile.ma41 .
Makefile.inc:../SRC/Makefile.$(ARCH)_$(PREC)
    cp -f ../SRC/Makefile.$(ARCH)_$(PREC) Makefile.inc

```

We have to specify, in the makefile, after 'PREC = ', the type required. In this case, it's 'SIMPLE'.

- Tape 'make', in the directory.

1.2 Used syntax

To generate the different versions, we have to change :

- Type of some variables (some variables become complex, some have to stay real);
- Part of some instructions;
- Name of subroutine from BLAS (not the same name according to the type of the variables).

In the two following notices, there is the transformations used in the code :

SRC	DBL	SIMPLE
DOUBLE PRECISION	DOUBLE PRECISION	REAL
DOUBLE PRECISION2	DOUBLE PRECISION	REAL
DOUBLE PRECISION3	DOUBLE PRECISION	DOUBLE PRECISION
DBLE	DBLE	REAL
DBLE2	DBLE	REAL

Notice for generating double an simple precision real.

SRC	CMPLX16	CMPLX
DOUBLE PRECISION	COMPLEX*16	COMPLEX
DOUBLE PRECISION2	DOUBLE PRECISION	REAL
DOUBLE PRECISION3	DOUBLE PRECISION	DOUBLE PRECISION
DBLE	DCMPLX	CMPLX
DBLE2	DBLE	REAL

Notice for generating double an simple precision complex.

Comment : In all versions **DOUBLE PRECISION3** becomes **DOUBLE PRECISION**, because of use of *MPI_WTIME* (is declared as **DOUBLE PRECISION** in *MPI*).

Remark : If in the *SRC* files, code was written for the double complex version, to generate the others, no modifications in the nome of different type (**DOUBLE PRECISION2**, **DBLE2**, ...) are necessary.

For the *MPI* variables, there is no change in the initial code.

1.3 Generation of the double precision real version from the *SRC* file

Here the script *clean_sgi_DBL.ex* changes nothing in the code (the type and the function name is the same). The only purpose is to clean the code. For example, if in a file located in the *SRC* directory, we have the declaration :

```
DOUBLE PRECISION2 RINFOG(20)
DOUBLE PRECISION A( LA ).
```

It becomes, in *DBL* :

```
DOUBLE PRECISION RINFOG(20)
DOUBLE PRECISION A( LA ).
```

Another example in *SRC*, we have the instruction :

```
ERMAX = ERMAX / DBLE2( N )
A(POSELT:LAPOS2) = DBLE(ZERO).
```

It becomes, in *DBL* :

```
ERMAX = ERMAX / DBLE( N )
A(POSELT:LAPOS2) = DBLE(ZERO).
```

1.4 Generation of the simple precision real version

This version is generated with the aid of the script *clean_sgi_SIMPLE.ex*. For example in *SRC*, we have the declaration :

```
DOUBLE PRECISION2 RINFOG(20)
DOUBLE PRECISION A( LA ).
```

It becomes, in *SIMPLE* :

```
REAL RINFOG(20)
REAL A( LA ).
```

Another example in *SRC*, we have the instruction :

```
ERMAX = ERMAX / DBLE2( N )
A(POSELT:LAPOS2) = DBLE(ZERO).
```

It becomes, in *SIMPLE* :

```
ERMAX = ERMAX / REAL( N )  
A(POSELT:LAPOS2) = REAL(ZERO).
```

For the function name (BLAS) : *DNRM2*, *DGEMV*, *DSCAL* ..., automatically become *SNRM2*, *SGEMV*, *SSCAL* The “D” id for “DOUBLE PRECISION REAL” and the “S” is for “SIMPLE PRECISION REAL”. This modification is also done by the script *clean_sgi_SIMPLE.ex*.

1.5 Generation of the simple precision complex version

This version is generated with the aid of the script *clean_sgi_cmplx.ex*. For example in *SRC*, we have the declaration :

```
DOUBLE PRECISION2 RINFOG(20)  
DOUBLE PRECISION A( LA ).
```

It becomes, in *CMPLX* :

```
REAL RINFOG(20)  
COMPLEX A( LA ).
```

Another example in *SRC*, we have the instruction :

```
ERMAX = ERMAX / DBLE2( N )  
A(POSELT:LAPOS2) = DBLE(ZERO).
```

It becomes, in *CMPLX* :

```
ERMAX = ERMAX / REAL( N )  
A(POSELT:LAPOS2) = CMPLX(ZERO).
```

Comment : In the two complex versions, only symmetric complex matrix are considered (hermitian matrix would require some work).

For the function name (BLAS) : *DNRM2*, *DGEMV*, *DSCAL* ..., become *SCNRM2*, *CGEMV*, *CSCAL* The “C” is for “SIMPLE PRECISION COMPLEX”. This change is performed with the script *clean_sgi_cmplx.ex*.

1.6 Generation of the double precision complex version

This version is generated with the aid of the script *clean_sgi_cmplx16.ex*.
For example in *SRC*, we have the declaration :

```
DOUBLE PRECISION2 RINFOG(20)
DOUBLE PRECISION A( LA ).
```

It becomes, in *CMPLX16* :

```
DOUBLE PRECISION RINFOG(20)
COMPLEX*16 A( LA ).
```

Another example in *SRC*, we have the instruction :

```
ERMAX = ERMAX / DBLE2( N )
A(POSELT:LAPOS2) = DBLE(ZERO).
```

It becomes, in *CMPLX16* :

```
ERMAX = ERMAX / DBLE( N )
A(POSELT:LAPOS2) = DCMPLX(ZERO).
```

For the function name (BLAS) : *DNRM2*, *DGEMV*, *DSCAL* ..., become *SZNRM2*, *ZGEMV*, *ZSCAL* The “Z” is for “DOUBLE PRECISION COMPLEX”. This change is performed with the script *clean_sgi_cmplx16.ex*.

2 Validation

2.1 Coverage

2.1.1 Interest of coverage

For this work, it was useful to have a tool which indicates the executed lines of a program.

One of the aim of the tester, is to “cover” a maximum number of line of the code. Such tool provides us with important informations. For example, *MUMPS* is composed of many files, and it is important to know how every file is used and which lines in each file are executed (or not executed). In this way, we can more easily create adapted tests and try to cover a maximum of the code.

For coverage of *MUMPS*, I have needed the such tool for fortran codes. I have used a tool called **FCAT** (**F**ortran **C**overage **A**nalysis **T**ool) introduced in next section.

2.1.2 Presentation of FCAT

FCAT was developed as part of software engineering process for a numerical analysis software project. FCAT is a recent tool, first version was created in January 2001.

FCAT is very convenient for those who are developing numerical software using fortran. It is designed to working mainly with f90/f95, even through it can also work with fixed formatted FORTRAN and was used in this context for MUMPS coverage analysis.

What it is able to do

FCAT holds two leading properties:

- find out “cold spot” in Fortran codes (the parts of the codes that are never executed), and flag these parts line-by-line.
- find out “hot spot” in Fortran codes (the parts of the codes that are most frequently executed), and give a line by line count.

What it is convenient in FCAT

- For software developers:
Finding out “cold spot” is particularly useful for code developers during the validation of their software in either eliminating these “cold spot”, or in designing more complete test suite to fully test these parts of the codes.
Finding “hot spot” could be useful for developers in optimizing their code.(For my work, I will use this aspect of FCAT).
- For software users:
Both capabilities also help a user in the analysis and understanding critical parts of the software used.

Using FCAT for larger codes

The examples given in this part are extracted from the tester of MUMPS. If you use a Makefile it is easy to incorporate FCAT usage into the Makefile.

- First, you need to create a tmp directory
- Secondly, you must add in Makefile the following lines:

```
fcats $ <> tmp/$ *.F &&\
$(FC) $(FFLAGS) -c tmp/$*.F
rm -f tmp/$*.F
```

1. The first line means each file `.F` is preprocessed by FCAT in the `tmp` directory.
 2. The following line corresponds to compile files in `tmp` directory and to create files `.mod`.
 3. The last line removes files in `tmp` directory.
- Finally, to obtain line coverage information you first execute your test program (here `test_mumps`) which generates an output file. Your output is made by a command like this:

```
mpirun -np Nbprocs test_mumps < input > output
```

where `Nbprocs` is the number of process, `input` the input file and `output` the output file. Note that both coverage information and “classical” program output are now contained in the output file. The form of the output file is explained in the following part (The main steps of FCAT with an example). This file is not easy to read, and a postprocessing is required to analyse it.

The postprocessing is done using the `fcats` command:

```
fcats test_mumps.F output > output1
```

where `output1` is the file obtained after postprocessing.

In our case, we want to analyse the coverage of the all files composing MUMPS library (323 subroutines), so it is necessary to write a script that analyses every source file. An example of script is present in the annex, it analyses every file and makes a synthesis to know how many lines are executed (or not executed) in each file.

The main steps of FCAT with an example

During the preprocessing, each file is directed in `tmp` directory, each executable line of a file is marked by a countenance like:

```
FCAT_name_of_file (number)
```

For example, first lines of `test_mumps.F` give:

```
call FCAT_test_mumps(1)
CALL MPI_INIT(IERR) .
call FCAT_test_mumps(2)
mumps_par%COMM = MPI_COMM_WORLD
call FCAT_test_mumps(3)
CALL MPI_COMM_RANK(mumps_par%COMM, MYID, IERR)
call FCAT_test_mumps(4)
IF (MYID .eq. MASTER) THEN
```

```

call FCAT_test_mumps(5)
NBtests1_1=0;NBtests1_2=0;NBtests1_3=0;NBtests1_4=0;

      :

call FCAT_test_mumps(9)
OPEN(UNIT=INFILE, FILE=DATA1, STATUS='OLD', ERR=500)
call FCAT_test_mumps(10)
REWIND INFILE
call FCAT_test_mumps(11)
OPEN(UNIT=INFILE_ELEM, FILE=DATA2, STATUS='OLD', ERR=500)
call FCAT_test_mumps(12)
REWIND INFILE_ELEM

```

At the end of this stage, files .mod are created. You then build an output file (called output in the previous part) containing all lines executed in the shape of FCAT_name_of_file (number).

For example a part of test_mumps output is:

```

FCAT_test_mumps_      1
FCAT_test_mumps_      1
FCAT_test_mumps_      1
FCAT_test_mumps_      2
FCAT_test_mumps_      3

      :

FCAT_test_mumps_      211
FCAT_test_mumps_      212
FCAT_test_mumps_      213

      :

FCAT_usedbytesthb_    169
FCAT_usedbytesthb_    170
FCAT_usedbytesthb_    171

      :

FCAT_psl_mumps_       28
FCAT_psl_mumps_       33
FCAT_psl_mumps_       34

```

```

FCAT_psl_ma41_ini_      1
FCAT_psl_ma41_ini_      1
FCAT_psl_ma41_ini_      2

```

The number *i* references the line in `test_mumps.F` (after preprocessing) “call `FCAT_test_mumps(i)`”. That means if the line following the sentence “call `FCAT_test_mumps(i)`” (in `test_mumps.F` after preprocessing) is executed then it is referenced in the output file.

In fact, during the postprocessing (`fcat test_mumps.F` output), FCAT compares files output and `test_mumps.F`. As soon as it finds in output file a same line like in `test_mumps.F`, it counts this line as a line executed.

If in the above examples, the line “`FCAT_test_mumps (1)`” is present on both files, therefore it will count the corresponding line in `test_mumps.F` executed.

Finally, after preprocessing, it marks in `test_mumps.F` with “`* >`” lines not executed:

For example :

```

      IF ((ALL_TEST.eq.0).or.(ALL_TEST.eq.1)) THEN
        GOTO 400
*>  ELSE
*>      write(OUT,*) 'The entered number is not correct'
*>  ENDIF

```

Options of FCAT

- dealing with fixedformat FORTRAN
You must add in preprocessing and postprocessing “-fixedform”
`fcat -fixedform ...`
- count option
You can have ahead of every line executed an integer count of the number of times this line has been executed.
You must add in preprocessing and postprocessing “-count” :
`fcat -count ...`
- report option
You can get a list of lines sorted by the number of times each line has been executed
You must add in preprocessing “-count”:
`fcat -count ...`

You must add in postprocessing “-count -report”:

```
fcatt -count -report...
```

Of course you may mix different options.

What I think about FCAT

It is a convenient tool but apparently it is difficult when you work with parallel process and when you start again to run a code with a different number of process, to add informations. For the big packages the output file can be very large and not very much readable.

The most important thing to know was how many and which lines are not executed and that was perfectly done by FCAT. The report option is not really useful for larger codes (MUMPS included) and the count option does not work correctly on large codes. I successfully run FCAT with this option on small code but I did not succeed on MUMPS.

After have presented the FCAT tool, I would like to explain the main part of my work, it consisted to do a tester for MUMPS.

2.2 Tester

2.2.1 Tester interest

MUMPS is a large code with a lot of options, and when we use this package often only a small part of MUMPS is used. Because of that, it was critical to create a tester containing many tests, each one concerned by a specific part of code. Thereby we can generate little by little a program that covers all part of the code. A tester is also convenient when we have a software that always changes. We can check at any moment if we do not have introduced a bug. I have then tried to make a structured tester that:

1. Validate the functionalities describe in the specification sheet
2. Easy to change
3. Easy to add new tests
4. Possible for the user to select/run a specific test and choose level of printing

2.2.2 General structure of the tester

The tester is composed of three parts:

1. Part 1 :
It corresponds to the main program. It mainly manages reading of matrix, call tests, informations given by users.
2. Part 2 :
It contains all test subroutines.
3. Part 3 :
Refers to all tools used by the previous part. It manages memory allocation, reading and building matrix, errors on the tests ...

Let us describe in more details each part of the tester.

- Part 1

- * Firstly, It reads the input files containing matrix on different formats and It generates a few random matrices.
- * Secondly, It asks the users for their choices about the kind of tests they want to execute. They have two possibilities :
to choice to do all tests or to choice to do a single test, in which case they must give complementary informations like number of test, type of the matrix, level of printing ...
- * Finally, It calls test subroutines with the informations below collected.

- Part 2

Let us give the general meaning of each class of tests:

- * Test of out of range values of input parameters
The general idea of test out of range is to see the behaviour of MUMPS when out of range input parameters are given by the user. That especially enable to see if error messages were provided and that they correspond the error made. Moreover, we have checked default parameters used by MUMPS.
This subroutine contains five other subroutines. Let us quickly look at them.
 - Test problem definition
It concerns parameters of the problem definition :(JOB, PAR, SYM)
 - Test matrix order
It concerns the order of input matrix (For example negative values are provided on input).
 - Test assembled matrix
It concerns parameters defining an assembled input matrix (NZ, IRN, JCN, RHS, A, COLSCA, ROWSCA)

- Test control parameters
It concerns most control parameters (ICNTL), (some of them are not considered).
 - Broadcasting test
Although this test is slightly different from others in this class of test, it was grouped to out of range test for the following reason.
In fact, it concerns parameter values, that have different values on each process. Only tests on parameters of problem definition are currently done but it should be easy to add tests on others parameters.
 - Test elemental matrix
It concerns parameters defining an elemental input matrix (NELT, ELTPTR, ELTVAR, A_ELT).
- * Printing test
This subroutine gathers together all that concerns readout of MUMPS (ICNTL (1 → 3)).
It is not executed when you run all tests because it is a test that has been built to understand better how work these three parameters.
- * Max trans test
It concerns the option of maximum transversal (ICNTL(6)).
This subroutine groups two other subroutines. According to the specification sheets I have separated symetric and unsymetric matrix cases. Then I have max_trans_sym_matrix and max_trans_unsym_matrix subroutines. They make nearly the same tests but of course have different results.
- * Memory test
This subroutine is designed to receive all tests concerning memory problems.
Only test on parameters MAXS and MAXIS are currently done.
It is interesting to notice that the tests are further subdivided in two parts:
- The host is not involved in facto/solve phases.
 - The host is involved in facto/solve phases.
- (Define by the value of PAR).
- * Test of ordering
It concerns the option of ordering (ICNTL(7))
- * Scaling test
It concerns the option of scaling strategy (ICNTL(8)).
In the same way as it has been done for the subroutine Max_trans_test, I have separated symetric and unsymetric matrix cases.

* Solve option test

It gathers together the control parameters accessible during the solve phase (ICNTL(9), ICNTL(10), ICNTL(11)).

* Schur test

It tests the option of schur complement (ICNTL(19)).

In fact, this subroutine only contains calls for subroutine used-schur1 (which is described in the next part).

• Part 3

This part groups all subroutines that are considered like tools.

Concerning the storage of matrices, it has seemed me necessary to create three different structures. Indeed the tester uses different sizes of matrix corresponding to the three structures. The first structure contains the small matrices (size <10), the second the random matrices (size >100), the third the random matrices (any size). It was not conceivable to create a single structure, that would have generated a memory loss.

Let us give the meaning of each subroutine.

* Read small matrix

It reads two input files: input_small_ass, input_small_elem.

Input_small_ass contains small matrix in assembled format while input_small_elem contains small matrix in elemental format. These two matrix types are stored in variables whose name ends by the character W.

It is very convenient to have the possibility to get back any matrix at all moment.

* Random matrix

It generates many assembled random matrix and stores them in variables whose name ends by the characters RW. The generation of random matrix is done with the subroutine YM01A.

* Random matrix input

It creates a single assembled random matrix whose parameters are entered by the user (N, NZ, SYM). It stores it in variable ended by RIW.

* Distrib matrix

If the matrix is in assembled format this subroutine manages the distribution (or not) of the matrix following the values of ICNTL(18).

* Alloc

It manages the memory allocation for the variables like (IRN, JCN, A, ...).

This is done for the two matrix formats.

- * Initialize an instance
It is involved on every test. It initializes an instance with JOB=-1 and stores the parameter values of a matrix in the mumps structure. It uses variables whose name ends by W, RW, IRW depending of the type of matrix to get back the wished matrix.
- * Clean1, clean2, clean3
There are three subroutines which manage the mumps structure deallocation and the arrays deallocation whose name ends by W, RW, IRW.
- * Usedbyschur1
Two tests are done, the first one checks the factorisation, the other verifies the accuracy of solution. For understand what it is really done, it must look at the code.
- * Usedbyschur2
It is only used by usedbyschur1. It generates right hand side especially designed to do the schur test validation.
- * Error
It manages all the error cases.
In general, it is called by each test, however there are tests for which the validation is rather complicated. The error analysis is then done in the test and there is no call to this subroutine.
It groups two subroutines: analysis_error, computation_error.
 - Analysis error
It analyses the possible errors that MUMPS may return. According to the test, we normally know the error generated by MUMPS, then we can compare between the expected answer and the answer generated by MUMPS.
 - Computation error
This subroutine only runs if the solve phase is completed (INFOG (1) equal zero after the solve phase). It checks that the calculated solution is acceptable. For that, we compute two errors:
 - . Relative error if we know the exact solution.
 - . Residual error in all cases.

A test is validated only if these two subroutines provide correct results.

For more informations you have to look at the code, that is well commented.

2.2.3 Numbering of tests and matrix

- Numbering of tests
It is necessary to have a numbering of tests where we can easily find

one's bearings. A test is defined by two or three numbers according to the tests. The tests are then numbered like this:

- The first level of numbering corresponds to the subroutines given in the description of part 2.
For example the subroutine "test_out_of_range" corresponds to number 1, "printing_test" number 2, ..., "schur_test" number 8. This number is performed by the variable N_CASE.
- In each subroutine, there is a second level of numbering. It either corresponds to a subroutine (for ex. "test_problem_definition" in "test_out_of_range") or it corresponds to a test (for ex. in "solve_option_test"). This number is performed by the variable N_CASE_LOC.
- Finally, it may exist a third level of numbering. This number is performed by the variable N_CASE_LOC_LOC.

Let us give examples of numbering (see also section B):

- The first test in the subroutine "test_problem_definition" is SYM=-10. This test is then defined by numbering 1.1.1 .
The first 1 means the test is contained in subroutine "test_out_of_range".
The second 1 means it is contained in subroutine "test_problem_definition"
The third 1 means it is the first test in subroutine "test_problem_definition"
- An other example where there is not a third level.
The subroutine "solve_option_test" corresponds to the number 7.
Its first test is ICNTL(9)=5. This test is then defined by the numbering 7.5 .

To have all the numbers of the tests you can look at the recapitulating array in rider.

- Numbering of matrix

The numbering is very simple. The tester is foreseen to run for this kind of numbering:

- Matrix numbered from 1 to 49 correspond to matrix in assembled format. They are in the file input_small_ass.
- Matrix numbered from 50 to 99 correspond to matrix in elemental format. They are in file input_small_elem.
- Matrix numbered from 100 to 999 correspond to random matrix. They are generated by the subroutine "random_matrix".
- The matrix with number 1000 is a random matrix which the parameters are provided by the user. It is generated by the subroutine "random_matrix_input".

2.2.4 Structure of a single test

Let us give for example the structure of the subroutine “test_problem_definition”, all tests are nearly built in the same way. We can decompose the structure of one test in four stages:

- Analysis of the kind of test to execute

In input of the subroutine we have the variables:

MATRIX_NUMBER, N_CASE_LOC, N_CASE_LOC_LOC

The presence of the variable MATRIX_NUMBER is essential whereas N_CASE_LOC and N_CASE_LOC_LOC are optional variables. If these two variables are present, we know we must do a single test otherwise we must do all tests contained in the subroutine. If a single test is required, it is verified N_CASE_LOC_LOC corresponds to an existing test. The validity of N_CASE and N_CASE_LOC is respectively checked in the main program and in subroutine “test_out_of_range”.

- Achievement of the test

We select the test to execute, we call MUMPS.

- Analysis of results obtained

In general, this is done by a call to the subroutine “error” but it is possible it is done in the subroutine containing the test.

- Desinitialization of the instance

If there was not any errors during call MUMPS with JOB=-1, this stage does a call to MUMPS with JOB=-2 and deallocate the structures.

If there is any interest and the matrix is assembled in every test is executed for the values 0-1-2-3 of ICNTL(18). Moreover, some tests can not be executed according to the number of processes or the value of PAR (PAR=0, Nprocs=1) and are not performed.

2.2.5 Using tester

- Run tests

When you run the tester, you have two possibilities: to do all tests or a single test. That is symbolized by the following sentence:

If you want to execute all tests tape 1 otherwise tape 0:

To do all tests tape 1 otherwise tape 0.

If you have reply 1 all tests are executed and you no more intervene. You obtain an output in this type:

TEST 1 , 1 , 1 , matrix number 1 , ICNTL(5)= 0 , ICNTL(18)= 0 IS OK

TEST 1 , 1 , 2 , matrix number 1 , ICNTL(5)= 0 , ICNTL(18)= 0 IS OK

TEST 1 , 1 , 3 , matrix number 1 , ICNTL(5)= 0 , ICNTL(18)= 0 IS OK

TEST 1 , 1 , 4 , matrix number 1 , ICNTL(5)= 0 , ICNTL(18)= 0 IS OK

At the end of the report you have the number of executed tests. The readout being standard, the value of ICNTL(18) for ICNTL(5)=1 may not have any meaning, for example when matrix is in element input format (ICNTL(5)=1).

If you have input 0, informations about the test are necessary:

- Firstly, the number of the test, in the shape of:

N_CASE :
N_CASE_LOC :
N_CASE_LOC_LOC :

- Secondly, matrix number.
- Thirdly, level of printing, it corresponds to the value of ICNTL(4).

Sometimes, complementary informations are requested from you:

If matrix number is 1000, you have to give the size of the matrix and the number of non zeros.

If the test is test of schur, you have to give the size of schur matrix.

At the end of the test you have the following informations:

- the solution (if the size of the matrix $N \leq 10$)
- the machine error
- Absolute error: the error with respect to the exact solution (if it is known)
- the residual ($\|b - Ax\|$) called ERRMAX in tester.
- the validation of the test

If it is validated, you have a line of this type:

*TEST 1 , 1 , 1 , matrix number 1 , ICNTL(5)= 0 ,
ICNTL(18)= 0 IS OK*

otherwise you have two possible error types:

- * MUMPS does not do what it was expected (specified in the specification sheets for example). It is characterized by the following sentence:

*Problem detected during TEST_OUT_OF_RANGE (1) test
call TEST_PROBLEM_DEFINITION (1) num 1 , matrix
number 1 , ICNTL(5) = 0 , ICNTL(18) = 0*

* MUMPS does not produce a good solution. It is characterized by the following sentence:

The calculated solution is bad in test 4 number 1 , 1 matrix number 1 , ICNTL(5) = 0 , ICNTL(18) = 0

- Add tests

If you want to add tests don't forget to do this:

- modify the number of the maximum test.

These are the following variables:

N_CASE_MAX, N_CASE_LOC_MAX (or NUM_LOC_MAX),

N_CASE_LOC_LOC_MAX (or NUM_LOC_LOC_MAX).

Following the cases, you have to change the number that allows the checking of the existence of the test. (Statement before the sentence "*THIS TEXT DOES NOT EXIT*").

- perhaps add this case in the error subroutines.

- Add matrix

In the same way it has been done for the tests don't eventually forget to change values of maximum (NMAX,NZMAX,...), values of NBmatrix_* in input files.

2.2.6 Results obtained with the tester

In this part we make a synthesis of results obtained. I will call by default value, the value used by MUMPS when an out of range value is given to an input parameter after the initialisation of MUMPS.

- Problem detected

- Warning mismanagement for IRN and JCN. If IRN or JCN have a minus number (or an out of range) there is a problem in subroutine "MUMPS_SET_INFOG" in file "PSL_MUMPS.F". It has been resolved by doing a modification in this subroutine.

- Problem with MAXS (size of the real workspace required for factorization and/or solve). Normally if the value of MAXS is less than INFO(8) (Minimum value of MAXS estimated by the analysis phase to run factorisation successfully) , MAXS takes the value of

INFO(8). It was not the case. This problem has been resolved by doing a modification in subroutine psl_ma41_analysis.F .

- This problem concerns the readout. If you give ICNTL(1) or ICNTL(2) or ICNTL(3) the value 5 there is a bug. The number 5 indeed represents the standard input and we can not write on it. This problem has not been resolved.
- If we give PAR a minus value , MUMPS treats it as zero from the first call to the subroutine “MUMPS_ID” (in file usedbyanalysis.F) , but if we run MUMPS on a single process, the checking of the incompatibility (in file psl_mumps.F) between PAR = 0 and NPROCS is done before the call to “MUMPS_ID” then there is a bug. It has been easily resolved by doing a modification in subroutine psl_mumps.F.

- Eventually to modify

There are eventually many things to modify which do not concern only the numerical aspect of MUMPS.

- If NZ (number of zeros) is greater than N^2 the dismissed error is INFOG(1) = - 22 and INFOG (2) =1, that is an error on IRN. I do not think it is the waited error. I believe the error INFOG(1)=-2 (NZ out of range) would be better suitable (Test 1.3.4).
- If we do not provide ICNTL(5) (matrix format) with the value 0 or 1, the value 1 is used by default. This value represents the matrix in elemental format. It seems better to use the value 0 by default moreover that is this value which is used if we do not provide any value on input.(Tests 1.4.4; 1.4.5)
- When the host does not work the values of INFO(7) and INFO(8) are set as zero. But the values of MAXS and MAXIS do not have the same values than INFO(7) and INFO(8), moreover MAXS and MAXIS are not treat alike. That do not have any consequence on the computation but it is not consistent with the general idea. (Tests 4.1.1; 4.1.2)
- In the analysis and factorization phases there is a report which says how the computation is worked out. This report is not provided after the solve phase. For example in test 7.2 (max steps iterative refinement set to 1), the value of INFOG(1) equal 8 with larger matrix (meaning :More than ICNTL(10) iterations are required) but this value is nowhere visible.

- Eventually to add in specifications

- The default value of SYM is 0 (Tests 1.1.1; 1.1.2; 1.1.3)
- The default value of PAR is 0 (Tests 1.1.4; 1.1.5; 1.1.6)

- If IRN or JCN have minus or out of bound value, it ignores them and only takes in account correct indices (Tests 1.3.5; 1.3.6; 1.3.8).
- If the size of the variable A is greater than NZ, it only computes with the NZ values. (Test 1.3.9)
- If ICNTL(10) has a minus value, it is treated as 0.
- ICNTL(4) has default value 2 if ICNTL(4) < 2, and 4 if ICNTL(4) >4. (Tests 1.4.2; 1.4.3)
- Default value of ICNTL(5).
- Scaling option is incompatible with Schur option.
- If ICNTL(11) is negative or zero it is treated as 0.

2.2.7 Results of coverage

At the end of my training, I have used FCAT to see if the coverage on MUMPS with the tester is satisfying. I have done the concatenation of two output files. The first is the result after the execution of MUMPS on a single process, the second the result after the execution of MUMPS on 24 process. The obtained coverage is in annex.

We then have a coverage nearly equal 55 %. This result is a little disappointing but I think the main part of tests concerning the errors can be introduced by the users have been done.

Moreover we can establish some files are not used, for example all subroutines concerning the root, usedbyQR.F (which is a larger file). If we do not count the large file usedbyQR.F we obtain a coverage nearly equal 61 % .

To run FCAT on MUMPS some file names must have been changed. The file name can not exceed a maximum number of characters, but FCAT during preprocessing adds the prefix FCAT so it increases the number of characters and can generate an error. It is an other drawback of FCAT.

Below there are the alterations made.

process_root_cont_static.F	⇒	process_root_cont.F
psl_mumps_free_block_cb.F	⇒	psl_mumps_free.F
process_last_rtnelind.F	⇒	process_last_rt.F
process_sym_bloc_facto.F	⇒	process_sym_bloc.F
LDLT_usedbyfacto_niv2.F	⇒	LDLT_usedbyfa_niv2.F
process_blfac_slave.F	⇒	process_blfa_slv.F
build_sort_index_ELT.F	⇒	build_sor_ind_ELT.F

3 Complex validation

Evaluation of the different versions by tester (only the tests with problems are indicated):

- Double real and complex :
 - Test 3.2.4 : infinity loop with use of big random matrix ($N > 26$) in the subroutine MC64RD.
 - Test 6.1.1 : problem detected with input 5 - 7 - 101 (ICNTL(5)=0, ICNTL(18)=0) and with input 51 (ICNTL(5)=1, ICNTL(18)=0).
- Simple real and complex :
 - Test 3.2.4 : infinity loop with use of big random matrix ($N > 26$) in the subroutine MC64RD.
 - Test 6.1.1 : problem detected with input 5 - 7 - 101 (ICNTL(5)=0, ICNTL(18)=0) and with input 51 (ICNTL(5)=1, ICNTL(18)=0).
 - Tests 3.2 - 4 - 5 - 6 : solution bad calculated.
 - Test 8 : problem detected with input 101 (random matrix) (ICNTL(18)=0, 1, 2, 3).

Conclusion

In the first part of this report, the generation of the different versions of MUMPS was presented. In these 4 versions, very few bugs are still to be corrected (but there are in particular cases as shown in the third part). In general, tests are running.

On the other hand, in the complex version, hermitian matrices are not considered and only symmetric complex and general unsymmetric complex matrices have been considered.

In a second part we have presented the tester and its functionalities, as well as the necessary tools to build a tester (like FCAT).

At the end of this part we have given the result of the obtained coverage. It is evident it still misses many tests to do, to have a full coverage. For example tests on the memory allocated can be interesting, moreover tests on the control parameters (ICNTL 12 -> 17) have not been performed. It is now possible to make adapted tests which take into account the current coverage obtained.

This training has been very formative since it has permitted us to discover new aspects of computing, like : using makefiles and debugger, shell, MPI ... We also have deepened our knowledge in Fortran 90.

Moreover, we were motivated by this job, because we knew it would be taken into account to improve MUMPS and would be used by other people.

Finally, being mixed with international researchers was very stimulating.

A Scripts for generation of different versions of MUMPS

A.1 Double precision real version

```
ed $1 <<!fined
g/^\$/d
g/^\ *$/d
g/INTEGER\*4/s/INTEGER\*4/INTEGER/
g/MPI\_INTEGER4/s/MPI\_INTEGER4/MPI\_INTEGER/
g/MUMPS[^\ ]* *(/s/MUMPS\[^\ ]*\) *(/MUMPS\1(/g
g/REAL\*8/s/REAL\*8/DOUBLE PRECISION/
g/DBLE2/s/DBLE2/DBLE/g
g/dble2/s/dble2/DBLE/g
g/DOUBLE PRECISION3/s/DOUBLE PRECISION3/DOUBLE PRECISION/g
g/DOUBLE PRECISION2/s/DOUBLE PRECISION2/DOUBLE PRECISION/g
g/double precision2/s/double precision2/DOUBLE PRECISION/g
g/FD05A /s/FD05A /FD05AD/g
g/FD05A(/s/FD05A(/FD05AD(/g
w
!fined
```

A.2 Single precision real version

```
ed $1 <<!fined
g/^\$/d
g/^\ *$/d
g/INTEGER\*4/s/INTEGER\*4/INTEGER/
g/MPI\_INTEGER4/s/MPI\_INTEGER4/MPI\_INTEGER/
g/MUMPS[^\ ]* *(/s/MUMPS\[^\ ]*\) *(/MUMPS\1(/g
g/DOUBLE PRECISION2/s/DOUBLE PRECISION2/REAL/g
g/double precision2/s/double precision2/REAL/g
g/DOUBLE PRECISION/s/DOUBLE PRECISION/REAL/g
g/double precision/s/double precision/REAL/g
g/REAL3/s/REAL3/DOUBLE PRECISION/g
g/MPI\_DOUBLE\_PRECISION/s/MPI\_DOUBLE\_PRECISION/MPI\_REAL/g
g/DBLE2/s/DBLE2/REAL/g
g/dble2/s/dble2/REAL/g
g/DBLE/s/DBLE/REAL/g
g/dble/s/dble/REAL/g
g/MPI\_2DOUBLE\_PRECISION/s/MPI\_2DOUBLE\_PRECISION/MPI\_2REAL/g
g/MPI\_REAL8/s/MPI\_REAL8/MPI\_REAL/g
g/[0-9]D[0-9]/s/\([0-9]\)D\([0-9]\)\1E\2/g
g/[0-9]d[0-9]/s/\([0-9]\)d\([0-9]\)\1E\2/g
```

g/[0-9]D-[0-9]/s/\([0-9]\)D-\([0-9]\)/\1E-\2/g
g/[0-9]d-[0-9]/s/\([0-9]\)d-\([0-9]\)/\1E-\2/g
g/[0-9]D+[0-9]/s/\([0-9]\)D+\([0-9]\)/\1E+\2/g
g/[0-9]d+[0-9]/s/\([0-9]\)d+\([0-9]\)/\1E+\2/g
g/DCOPY/s/DCOPY/SCOPY/g
g/PDDOT/s/PDDOT/PSDOT/g
g/DDOT/s/DDOT/SDOT/g
g/DSYR/s/DSYR/SSYR/g
g/PDSCAL/s/PDSCAL/PSSCAL/g
g/DSCAL/s/DSCAL/SSCAL/g
g/DGEMM/s/DGEMM/SGEMM/g
g/DSWAP/s/DSWAP/SSWAP/g
g/PDTRSM/s/PDTRSM/PSTRSM/g
g/DTRSM/s/DTRSM/STRSM/g
g/DGEMV/s/DGEMV/SGEMV/g
g/DGER/s/DGER/SGER/g
g/DTRSV/s/DTRSV/STRSV/g
g/PDTRTRS/s/PDTRTRS/PSTRTRS/g
g/DTRTRS/s/DTRTRS/STRTRS/g
g/PDGETRF/s/PDGETRF/PSGETRF/g
g/DGETRF/s/DGETRF/SGETRF/g
g/PDORMQR/s/PDORMQR/PSORMQR/g
g/DORMQR/s/DORMQR/SORMQR/g
g/PDGETRS/s/PDGETRS/PSGETRS/g
g/DGETRS/s/DGETRS/SGETRS/g
g/DPOTRS/s/DPOTRS/SPOTRS/g
g/DGEBS2D/s/DGEBS2D/SGEBS2D/g
g/DLASWP/s/DLASWP/SLASWP/g
g/DLARFG/s/DLARFG/SLARFG/g
g/DLARF/s/DLARF/SLARF/g
g/PDAXPY/s/PDAXPY/PSAXPY/g
g/DAXPY/s/DAXPY/SAXPY/g
g/DORM2R/s/DORM2R/SORM2R/g
g/IDAMAX/s/IDAMAX/ISAMAX/g
g/DLAMCH/s/DLAMCH/SLAMCH/g
g/PDNRM2/s/PDNRM2/PSNRM2/g
g/DNRM2/s/DNRM2/SNRM2/g
g/PDGEQPF/s/PDGEQPF/PSGEQPF/g
g/DGEQPF/s/DGEQPF/SGEQPF/g
g/PDPOTRF/s/PDPOTRF/PSPOTRF/g
g/DPOTRF/s/DPOTRF/SPOTRF/g
g/PDPOTRS/s/PDPOTRS/PSPOTRS/g
g/DPOTRS/s/DPOTRS/SPOTRS/g
g/DGEQR2/s/DGEQR2/SGEQR2/g

```

g/PDGECON/s/PDGECON/PSGECON/g
g/DGECON/s/DGECON/SGECON/g
g/PDLAPIV/s/PDLAPIV/PSLAPIV/g
g/DLAPIV/s/DLAPIV/SLAPIV/g
g/KEEP(35) = 8/s/KEEP(35) = 8/KEEP(35) = 4/g
g/REAL\*8/s/REAL\*8/REAL/g
g/FD05AD/s/FD05AD/FD05AS/g
g/FD05A /s/FD05A /FD05AS/g
g/FD05A(/s/FD05A(/FD05AS(/g
w
!fined

```

A.3 Single precision complex version

```

ed $1 <<!fined
g/^$/d
g/^ *$/d
g/INTEGER\*4/s/INTEGER\*4/INTEGER/
g/MPI\_INTEGER4/s/MPI\_INTEGER4/MPI\_INTEGER/
g/MUMPS[^ ]* *(/s/MUMPS\[^\ ]*\ ) *(/MUMPS\1(/g
g/DOUBLE PRECISION2/s/DOUBLE PRECISION2/REAL/g
g/double precision2/s/double precision2/REAL/g
g/DOUBLE PRECISION/s/DOUBLE PRECISION/COMPLEX/g
g/double precision/s/double precision/COMPLEX/g
g/COMPLEX3/s/COMPLEX3/DOUBLE PRECISION/g
g/MPI\_DOUBLE\_PRECISION/s/MPI\_DOUBLE\_PRECISION/MPI\_COMPLEX/g
g/MPI\_2DOUBLE\_PRECISION/s/MPI\_2DOUBLE\_PRECISION/MPI\_2REAL/g
g/MPI\_REAL8/s/MPI\_REAL8/MPI\_REAL/g
g/[0-9]D[0-9]/s/\([0-9]\)D\([0-9]\)/\1E\2/g
g/[0-9]d[0-9]/s/\([0-9]\)d\([0-9]\)/\1E\2/g
g/[0-9]D-[0-9]/s/\([0-9]\)D-\([0-9]\)/\1E-\2/g
g/[0-9]d-[0-9]/s/\([0-9]\)d-\([0-9]\)/\1E-\2/g
g/[0-9]D+[0-9]/s/\([0-9]\)D+\([0-9]\)/\1E+\2/g
g/[0-9]d+[0-9]/s/\([0-9]\)d+\([0-9]\)/\1E+\2/g
g/FD05AD/s/FD05AD/FD05AC/g
g/FD05A /s/FD05A /FD05AC/g
g/FD05A(/s/FD05A(/FD05AC(/g
g/DCOPY/s/DCOPY/CCOPY/g
g/PDDOT/s/PDDOT/PSDOT/g
g/DDOT/s/DDOT/SDOT/g
g/DSYR/s/DSYR/CSYR/g
g/PDSCAL/s/PDSCAL/PCSCAL/g
g/DSCAL/s/DSCAL/CSCAL/g
g/DGEMM/s/DGEMM/CGEMM/g

```

```

g/DSWAP/s/DSWAP/CSWAP/g
g/PDTRSM/s/PDTRSM/PCTRSM/g
g/DTRSM/s/DTRSM/CTRSM/g
g/DGEMV/s/DGEMV/CGEMV/g
g/DGER/s/DGER/CGERU/g
g/DTRSV/s/DTRSV/CTRSV/g
g/PDTRTRS/s/PDTRTRS/PCTRTRS/g
g/DTRTRS/s/DTRTRS/CTRTRS/g
g/PDGETRF/s/PDGETRF/PCGETRF/g
g/DGETRF/s/DGETRF/CGETRF/g
g/PDORMQR/s/PDORMQR/PCUNMQR/g
g/DORMQR/s/DORMQR/CUNMQR/g
g/PDGETRS/s/PDGETRS/PCGETRS/g
g/DGETRS/s/DGETRS/CGETRS/g
g/DGEBS2D/s/DGEBS2D/CGEBS2D/g
g/DLASWP/s/DLASWP/CLASWP/g
g/DLARFG/s/DLARFG/CLARFG/g
g/DLARF/s/DLARF/CLARF/g
g/PDAXPY/s/PDAXPY/PCAXPY/g
g/DAXPY/s/DAXPY/CAXPY/g
g/DORM2R/s/DORM2R/CUNM2R/g
g/IDAMAX/s/IDAMAX/ICAMAX/g
g/DLAMCH/s/DLAMCH/SLAMCH/g
g/PDNRM2/s/PDNRM2/PSCNRM2/g
g/DNRM2/s/DNRM2/SCNRM2/g
g/PDGEQPF/s/PDGEQPF/PCGEQPF/g
g/DGEQPF/s/DGEQPF/CGEQPF/g
g/PDPOTRF/s/PDPOTRF/PCPOTRF/g
g/DPOTRF/s/DPOTRF/CPOTRF/g
g/PDPOTRS/s/PDPOTRS/PCPOTRS/g
g/DPOTRS/s/DPOTRS/CPOTRS/g
g/DGEQR2/s/DGEQR2/CGEQR2/g
g/PDGECON/s/PDGECON/PCGECON/g
g/DGECON/s/DGECON/CGECON/g
g/PDLAPIV/s/PDLAPIV/PCLAPIV/g
g/DLAPIV/s/DLAPIV/CLAPIV/g
g/KEEP(35) = 8/s/KEEP(35) = 8/KEEP(35) = 8/g
g/REAL\*8/s/REAL\*8/REAL/g
g/DBLE2/s/DBLE2/REAL/g
g/dble2/s/dble2/REAL/g
g/DBLE/s/DBLE/CMPLX/g
g/dble/s/dble/CMPLX/g
w
!fined

```

A.4 Double precision complex version

```
ed $1 <<!fined
g/^\$/d
g/^\ */d
g/INTEGER\*4/s/INTEGER\*4/INTEGER/
g/MPI\_INTEGER4/s/MPI\_INTEGER4/MPI\_INTEGER/
g/MUMPS[^\ ]* *(/s/MUMPS\[^\ ]*\ ) *(/MUMPS\1(/g
g/MPI_REAL8/s/MPI_REAL8/MPI_REAL/g
g/DOUBLE PRECISION/s/DOUBLE PRECISION/COMPLEX\*16/g
g/double precision/s/double precision/COMPLEX\*16/g
g/COMPLEX\*162/s/COMPLEX\*162/DOUBLE PRECISION/g
g/COMPLEX\*163/s/COMPLEX\*163/DOUBLE PRECISION/g
g/MPI_2DOUBLE_PRECISION/s/MPI_2DOUBLE_PRECISION/MPI_2DOUBLE_PRECISION/g
g/MPI_DOUBLE_PRECISION/s/MPI_DOUBLE_PRECISION/MPI_DOUBLE_COMPLEX/g
g/REAL\*8/s/REAL\*8/DOUBLE PRECISION/g
g/DBLE/s/DBLE/DCMPLX/g
g/dble/s/dble/DCMPLX/g
g/DCMPLX2/s/DCMPLX2/DBLE/g
g/FD05AD/s/FD05AD/FD05AZ/g
g/FD05A /s/FD05A /FD05AZ/g
g/FD05A (/s/FD05A (/FD05AZ (/g
g/DCOPY/s/DCOPY/ZCOPY/g
g/PDDOT/s/PDDOT/PDDOT/g
g/DDOT/s/DDOT/DDOT/g
g/DSYR/s/DSYR/ZSYR/g
g/PDSCAL/s/PDSCAL/PZSCAL/g
g/DSCAL/s/DSCAL/ZSCAL/g
g/DGEMM/s/DGEMM/ZGEMM/g
g/DSWAP/s/DSWAP/ZSWAP/g
g/PDTRSM/s/PDTRSM/PZTRSM/g
g/DTRSM/s/DTRSM/ZTRSM/g
g/DGEMV/s/DGEMV/ZGEMV/g
g/DGER/s/DGER/ZGERU/g
g/DTRSV/s/DTRSV/ZTRSV/g
g/PDTRTRS/s/PDTRTRS/PZTRTRS/g
g/DTRTRS/s/DTRTRS/ZTRTRS/g
g/PDGETRF/s/PDGETRF/PZGETRF/g
g/DGETRF/s/DGETRF/ZGETRF/g
g/PDORMQR/s/PDORMQR/PZUNMQR/g
g/DORMQR/s/DORMQR/ZUNMQR/g
g/PDGETRS/s/PDGETRS/PZGETRS/g
g/DGETRS/s/DGETRS/ZGETRS/g
g/DGEBS2D/s/DGEBS2D/ZGEBS2D/g
```

```
g/DLASWP/s/DLASWP/ZLASWP/g
g/DLARFG/s/DLARFG/ZLARFG/g
g/DLARF/s/DLARF/ZLARF/g
g/PDAXPY/s/PDAXPY/PZAXPY/g
g/DAXPY/s/DAXPY/ZAXPY/g
g/DORM2R/s/DORM2R/ZUNM2R/g
g/IDAMAX/s/IDAMAX/IZAMAX/g
g/DLAMCH/s/DLAMCH/DLAMCH/g
g/PDNRM2/s/PDNRM2/PDZNRM2/g
g/DNRM2/s/DNRM2/DZNRM2/g
g/PDGEQPF/s/PDGEQPF/PZGEQPF/g
g/DGEQPF/s/DGEQPF/ZGEQPF/g
g/PDPOTRF/s/PDPOTRF/PZPOTRF/g
g/DPOTRF/s/DPOTRF/ZPOTRF/g
g/PDPOTRS/s/PDPOTRS/PZPOTRS/g
g/DPOTRS/s/DPOTRS/ZPOTRS/g
g/DGEQR2/s/DGEQR2/ZGEQR2/g
g/PDGECON/s/PDGECON/PZGECON/g
g/DGECON/s/DGECON/ZGECON/g
g/PDLAPIV/s/PDLAPIV/PZLAPIV/g
g/DLAPIV/s/DLAPIV/ZLAPIV/g
g/KEEP(35) = 8/s/KEEP(35) = 8/KEEP(35) = 16/g
w
!fined
```

B Numbering of tests

```
*****  
*  
*   N_CASE = 1      : TEST_OUT_OF_RANGE *  
*  
*****
```

N_CASE_LOC 1 TEST_PROBLEM_DEFINITION

N_CASE_LOC_LOC

1	SYM=	-10	
2	SYM=	-1	
3	SYM=	3	
4	PAR=	-1	
5	PAR=	-5	
6	PAR=	2	
7	JOB=	-3	replace JOB = -1
8	JOB=	-1	called two times
9	JOB=	7	replace JOB = -1
10	JOB=	2	jump the analysis phase
11	JOB=	-3	replace JOB = 6
12	JOB=	-5	replace JOB = 6
13	JOB=	7	replace JOB = 6
14	JOB=	1+2+3	replace JOB = 6

N_CASE_LOC 2 TEST_MATRIX_ORDER

N_CASE_LOC_LOC

1	N=	-1
2	N=	0
3	N=	-99

N_CASE_LOC 3 TEST_ASSEMBLED_MATRIX

N_CASE_LOC_LOC

```
1      NZ= -1
2      NZ= -10
3      NZ= 0
4      NZ= N*N+1
5      size IRN > size NZ
6      size IRN = size JCN > NZ
7      size IRN = size JCN = NZ-1
8      IRN(1)= -1
9      order A > NZ
10     order A < NZ
11     size RHS > order A
12     size RHS < order A
13     size JCN < NZ
14     JCN empty
15     A empty
16     size ROWSCA < N
17     COLSCA empty
18     size COLSCA < N
19     RHS empty
```

N_CASE_LOC 4

TEST_CONTROL_PARAMETER

N_CASE_LOC_LOC

```
1      ICNTL(4)= 0
2      ICNTL(4)= 5
3      ICNTL(4)= -1
4      ICNTL(5)= 2
5      ICNTL(5)= -1
6      ICNTL(6)= -1
7      ICNTL(6)= 7
8      ICNTL(7)= 2
9      ICNTL(7)= -1
10     ICNTL(8)= -2
11     ICNTL(8)= 9
12     ICNTL(10)= -4
13     ICNTL(14)= -10
14     ICNTL(14)=110
15     ICNTL(16)=-4
16     ICNTL(16)=11
```

```

17      ICNTL(19)=-1,SIZE_SCHUR=3, LISTVAR_SCHUR=(/N-1,N-2,N-3/)
18      ICNTL(19)=-20 ,SIZE_SCHUR=N+1,LISTVAR_SCHUR=(/1:N+1/)
19      ICNTL(19)=-2 ,SIZE_SCHUR=2,LISTVAR_SCHUR=(/1/)
20      ICNTL(19)=0 ,SIZE_SCHUR=2,LISTVAR_SCHUR=(/1,2/)
21      ICNTL(20)=1

```

```

N_CASE_LOC      5          BROADCASTING_TEST

```

```

N_CASE_LOC_LOC

```

```

1      JOB HAS NOT THE SAME VALUES ON EACH PROC
2      SYM HAS NOT THE SAME VALUES ON EACH PROC
3      PAR HAS NOT THE SAME VALUES ON EACH PROC

```

```

N_CASE_LOC      6          TEST_ELEMENTAL_MATRIX

```

```

N_CASE_LOC_LOC

```

```

1      NELT = -3
2      ELTPT empty
3      size ELTPTR < NELT+1
4      ELTVAR empty
5      size ELTVAR < LELTVAR
6      A_ELT empty
7      size A_ELT < NA_ELT

```

```

=====
=====

```

```

*****
*
*      N_CASE = 2      : PRINTING_TEST      *
*
*****

```

```

N_CASE_LOC

```

```

1      ICNTL(1)= -1
2      ICNTL(1)= 0
3      ICNTL(1)= 10

```

```

4      ICNTL(1)= 1
5      ICNTL(1)= 5
5      ICNTL(1)= 6
6      ICNTL(2)=-1
7      ICNTL(2)=0
8      ICNTL(2)= 10
9      ICNTL(2)= 1
10     ICNTL(2)= 5
11     ICNTL(2)= 6
12     ICNTL(2)= -1
13     ICNTL(3)=-1
14     ICNTL(3)=0
15     ICNTL(3)=10
16     ICNTL(3)=1
17     ICNTL(3)=5
18     ICNTL(3)=6

```

```

=====
=====

```

```

*****
*
*      N_CASE = 3      : MAX_TRANS_TEST      *
*                      (ICNTL(6))          *
*****

```

```

N_CASE_LOC  1      with symetric matrix

```

```

N_CASE_LOC_LOC

```

```

1      ICNTL(6)= 0
2      ICNTL(6)= 1
3      ICNTL(6)= 2
4      ICNTL(6)= 3
5      ICNTL(6)= 4
6      ICNTL(6)= 5
7      ICNTL(6)= 6
8      ICNTL(6)= 5 , ICNTL(18)=1, ICNTL(18)=2, ICNTL(18)=3

```

```

N_CASE_LOC  2      with unsymetric matrix

```

```

N_CASE_LOC_LOC
  1          ICNTL(6)= 0
  2          ICNTL(6)= 1
  3          ICNTL(6)= 2
  4          ICNTL(6)= 3
  5          ICNTL(6)= 4
  6          ICNTL(6)= 5
  7          ICNTL(6)= 6
  8          ICNTL(6)= 4 ICNTL(7)=1  (ORDERING)
9  ICNTL(6)= 3 ICNTL(19)=1 (SCHUR)
 10         ICNTL(6)= 5 , ICNTL(18)=1, ICNTL(18)=2, ICNTL(18)=3

```

```

=====
=====

```

```

*****
*
*   N_CASE = 4      : MEMORIES_TEST
*
*
*****

```

N_CASE_LOC 1 PAR=0 the host is not involved in factorization/solve phases

```

N_CASE_LOC_LOC
  1          MAXIS < INFO(7) ,MAXS <INFO(8)
  2          MAXIS > INFO(7) ,MAXS >INFO(8)

```

N_CASE_LOC 2 PAR=1 the host is involved in factorization/solve phases

```

N_CASE_LOC_LOC
  1          MAXIS < INFO(7) ,MAXS <INFO(8)
  2          MAXIS > INFO(7) ,MAXS >INFO(8)

```

```

=====
=====

```

```

*****
*
*   N_CASE = 5      : TEST_OF_ORDERING
*
*
*****

```

N_CASE_LOC

```

1      ICNTL(7)= 1      PERM_IN=(/(mumps_par%N-I,I=0,mumps_par%N-1)/)
2      ICNTL(7)= 1      PERM_IN=(/(mumps_par%N-I,I=1,mumps_par%N-1)/)
3      ICNTL(7)= 1      PERM_IN=(/(mumps_par%N-I,I=0,mumps_par%N)/)

```

```

=====
=====

```

```

*****
*
*   N_CASE = 6      : SCALING_TEST
*
*
*****

```

N_CASE_LOC 1 with symetric matrix

N_CASE_LOC_LOC

```

1      ICNTL(8)= -1 before the analysis; ICNTL(6)=5
2      ICNTL(8)= -1 before the analysis; ICNTL(6)=6
3      ICNTL(8)= -1 after the analysis; ICNTL(6)= 5
4      ICNTL(8)= 0
5      ICNTL(8)= 1
6      ICNTL(8)= 2
7      ICNTL(8)= 3
8      ICNTL(8)= 4
9      ICNTL(8)= 5
10     ICNTL(8)= 6
11     ICNTL(8)= 7
12     ICNTL(8)= -1 ROWSCA=RANDOM,COLSCA=RANDOM
13     ICNTL(8)= 3, SCHUR

```

```

14          ICNTL(8)= 3, ICNTL(18)=1, ICNTL(18)=2, ICNTL(18)=3

N_CASE_LOC      2          with unsymmetric matrix

N_CASE_LOC_LOC
  1          ICNTL(8)= -1 before the analysis; ICNTL(6)=5
  2          ICNTL(8)= -1 before the analysis; ICNTL(6)=6
  3          ICNTL(8)= -1 after the analysis; ICNTL(6)= 5
  4          ICNTL(8)= 0
  5          ICNTL(8)= 1
  6          ICNTL(8)= 2
  7          ICNTL(8)= 3
  8          ICNTL(8)= 4
  9          ICNTL(8)= 5
 10          ICNTL(8)= 6
 11          ICNTL(8)= 7
 12          ICNTL(8)= -1 ROWSCA=RANDOM, COLSCA=RANDOM
 13          ICNTL(8)= 3, SCHUR
 14          ICNTL(8)= 3, ICNTL(18)=1, ICNTL(18)=2, ICNTL(18)=3

```

```

=====
=====

```

```

*****
*                                     *
*   N_CASE = 7       : SOLVE_OPTION_TEST *
*                                     *
*****

```

```

N_CASE_LOC

  1          ICNTL(10)= 1
  2          ICNTL(10)= 5
  3          ICNTL(10)= 100
  4          ICNTL(11)= 0
  5          ICNTL(11)= 5
  6          ICNTL(11)= 100

```

=====
=====

```
*****  
*                                     *  
*   N_CASE = 8       : SCHUR_TEST     *  
*                   (in usedbyschur1) *  
*****
```

N_CASE_LOC

```
1          ICNTL(19)= 1   checking of facto phase  
2          ICNTL(19)= 1   checking of solve phase
```

=====
=====

C Analysis coverage

C.1 Example of a script using FCAT for a larger code

```
#!/bin/sh
dat='date'
echo " nom output = $1"
##for f in assemblies
echo " Bilan de coverage ${dat} " >> COV_${dat}
echo " ----- " >> COV_${dat}
for f in `ls *.F | sed -e 's/.F//g'`
do
  echo $f
  fcat -fixedform ${f}.F $1 > ${f}.cov 2>&1
  grep " Code .* has .* lines:" ${f}.cov >> COV_${dat}
  grep " are executable lines" ${f}.cov >> COV_${dat}
  grep " of which [0-9]* are not executed" ${f}.cov >> COV_${dat}
  echo " -- " >> COV_${dat}
  \rm ${f}.cov
##  fcat -fixedform ${f}.F $1 > ${f}.cov
done
ed COV_${dat} <<!fined
g/are executable lines/s/are executable lines/lignes executees/g
w
!fined
```

C.2 Result of the coverage

Bilan de coverage Fri Aug 10 13:26:28 CEST 2001

```
-----
Code LDLT_facto_niv1.F has 268 lines:
60 are executable lines
of which 30 are not executed
--
Code LDLT_facto_niv2.F has 339 lines:
74 are executable lines
of which 39 are not executed
--
Code LDLT_mpima41md.F has 730 lines:
328 are executable lines
of which 98 are not executed
--
Code LDLT_usedbyfa_niv2.F has 470 lines:
132 are executable lines
of which 30 are not executed
```

```
--
Code LDLT_usedbyfacto.F has 729 lines:
212 are executable lines
of which 22 are not executed
--
Code arrowheads.F has 1403 lines:
418 are executable lines
of which 119 are not executed
--
Code assemblies.F has 2966 lines:
1099 are executable lines
of which 363 are not executed
--
Code assemblies_ELTF has 2641 lines:
941 are executable lines
of which 782 are not executed
--
Code build_index.F has 312 lines:
156 are executable lines
of which 33 are not executed
--
Code build_index_ELTF has 291 lines:
138 are executable lines
of which 93 are not executed
--
Code build_sor_ind_ELTF has 614 lines:
236 are executable lines
of which 118 are not executed
--
Code build_sort_index.F has 722 lines:
278 are executable lines
of which 51 are not executed
--
Code cv_candidate.F has 664 lines:
350 are executable lines
of which 80 are not executed
--
Code distentry.F has 1209 lines:
434 are executable lines
of which 155 are not executed
--
Code elementdistrib.F has 1024 lines:
379 are executable lines
of which 190 are not executed
```

```
--
Code elemententry.F has 1887 lines:
638 are executable lines
of which 130 are not executed
--
Code estim_flops.F has 112 lines:
36 are executable lines
of which 0 are not executed
--
Code facto_niv1.F has 226 lines:
59 are executable lines
of which 17 are not executed
--
Code facto_niv2.F has 310 lines:
53 are executable lines
of which 11 are not executed
--
Code fd05ad.F has 96 lines:
8 are executable lines
of which 3 are not executed
--
Code ma41_propinfo.F has 98 lines:
16 are executable lines
of which 7 are not executed
--
Code mc64ad.F has 2002 lines:
1067 are executable lines
of which 199 are not executed
--
Code mpi_mcast2.F has 70 lines:
13 are executable lines
of which 4 are not executed
--
Code mpima41b.F has 307 lines:
38 are executable lines
of which 4 are not executed
--
Code mpima41c.F has 394 lines:
90 are executable lines
of which 28 are not executed
--
Code mpima41md.F has 765 lines:
433 are executable lines
of which 167 are not executed
```

```
--
Code mpima41r.F has 318 lines:
96 are executable lines
of which 41 are not executed
--
Code mpima41s.F has 1133 lines:
492 are executable lines
of which 121 are not executed
--
Code mpimc51d.F has 528 lines:
121 are executable lines
of which 61 are not executed
--
Code mpimc51e_niv12.F has 1347 lines:
324 are executable lines
of which 138 are not executed
--
Code mpimc51g_niv12.F has 882 lines:
260 are executable lines
of which 97 are not executed
--
Code mumps_bloc2.F has 1198 lines:
413 are executable lines
of which 220 are not executed
--
Code mumps_buffer.F has 3628 lines:
834 are executable lines
of which 324 are not executed
--
Code mumps_cv_load.F has 580 lines:
244 are executable lines
of which 104 are not executed
--
Code mumps_load.F has 430 lines:
112 are executable lines
of which 112 are not executed
--
Code mumps_root_facto.F has 2186 lines:
1051 are executable lines
of which 1051 are not executed
--
Code mumps_root_solve.F has 594 lines:
255 are executable lines
of which 255 are not executed
```

```
--
Code mumps_seq_root.F has 1547 lines:
738 are executable lines
of which 738 are not executed
--
Code process_bande.F has 194 lines:
44 are executable lines
of which 3 are not executed
--
Code process_blfa_slv.F has 550 lines:
146 are executable lines
of which 63 are not executed
--
Code process_blocfacto.F has 500 lines:
121 are executable lines
of which 50 are not executed
--
Code process_contrib.F has 303 lines:
81 are executable lines
of which 26 are not executed
--
Code process_factor.F has 122 lines:
38 are executable lines
of which 38 are not executed
--
Code process_last_rt.F has 302 lines:
76 are executable lines
of which 76 are not executed
--
Code process_maitre2.F has 250 lines:
50 are executable lines
of which 8 are not executed
--
Code process_maplig.F has 1267 lines:
370 are executable lines
of which 164 are not executed
--
Code process_node.F has 163 lines:
23 are executable lines
of which 0 are not executed
--
Code process_root2slave.F has 323 lines:
94 are executable lines
of which 94 are not executed
```

```
--
Code process_root2son.F has 523 lines:
166 are executable lines
of which 166 are not executed
--
Code process_root_cont.F has 173 lines:
41 are executable lines
of which 41 are not executed
--
Code process_rtnelind.F has 198 lines:
47 are executable lines
of which 47 are not executed
--
Code process_sym_bloc.F has 817 lines:
223 are executable lines
of which 95 are not executed
--
Code psl_ma41_analysis.F has 2059 lines:
773 are executable lines
of which 266 are not executed
--
Code psl_ma41_end.F has 260 lines:
73 are executable lines
of which 9 are not executed
--
Code psl_ma41_facto.F has 1877 lines:
626 are executable lines
of which 186 are not executed
--
Code psl_ma41_ini.F has 171 lines:
78 are executable lines
of which 7 are not executed
--
Code psl_ma41_solve.F has 1353 lines:
604 are executable lines
of which 180 are not executed
--
Code psl_mumps.F has 1414 lines:
521 are executable lines
of which 61 are not executed
--
Code psl_mumps_free.F has 106 lines:
50 are executable lines
of which 50 are not executed
```

```
--
Code root_level3.F has 2019 lines:
581 are executable lines
of which 581 are not executed
--
Code symmetrize.F has 360 lines:
93 are executable lines
of which 93 are not executed
--
Code tools.F has 1711 lines:
635 are executable lines
of which 256 are not executed
--
Code traiter_message.F has 1235 lines:
219 are executable lines
of which 138 are not executed
--
Code usedbyLDLT.F has 383 lines:
130 are executable lines
of which 35 are not executed
--
Code usedbyQR.F has 6686 lines:
2324 are executable lines
of which 2324 are not executed
--
Code usedbyanalysis.F has 9233 lines:
3037 are executable lines
of which 562 are not executed
--
Code usedbyfacto.F has 2464 lines:
1027 are executable lines
of which 483 are not executed
--
Code usedbyfacto_niv2.F has 383 lines:
112 are executable lines
of which 22 are not executed
--
Code usedbypima41r.F has 1186 lines:
422 are executable lines
of which 108 are not executed
--
Code usedbypima41s.F has 612 lines:
188 are executable lines
of which 64 are not executed
```

```
--  
Code usedbysolve.F has 1404 lines:  
534 are executable lines  
of which 184 are not executed  
--
```

References

- [1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17:886–905, 1996.
- [2] P. R. Amestoy, I. S. Duff, and J.-Y. L’Excellent. Mumps multifrontal massively parallel solver version 2.0. Technical Report RT/APO/98/3, ENSEEIHT-IRIT, Toulouse, France, 1998.
- [3] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee, 1995.
- [4] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report LAPACK Working Note 100, CS-95-283, University of Tennessee, 1995.
- [5] J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. MPI : A message passing interface standard. *Int. Journal of Supercomputer Applications*, 8:(3/4), 1995.
- [6] J. Dongarra and R. C. Whaley. LAPACK Working Note 94: A Users’ Guide to the BLACS v1.0. Technical Report UT-CS-95-281, University of Tennessee, Knoxville, Tennessee, USA, 1995.
- [7] PARASOL. Deliverable 2.1c: MUMPS Version 3.1. A MULTifrontal Massively Parallel Solver. Technical report, February 19, 1999.
- [8] PARASOL. Deliverable 2.1a: MUMPS Version 2.0. A MULTifrontal Massively Parallel Solver. Technical report, January 10, 1998.
- [9] PARASOL. Deliverable 2.1d (final report): MUMPS Version 4.0. A MULTifrontal Massively Parallel Solver. Technical report, June 30, 1999.
- [10] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. MPI: The Complete Reference. The MIT Press, Cambridge, Massachusetts, 1996.