qr_mumps Users Guide

2020 2020

Contents

1 Introduction

`qr_mumps` is a software package for the solution of sparse, linear systems on multicore computers. It implements a direct solution method based on the QR of Cholesky factorization of the input matrix. It is suited to solving sparse **least-squares** problems $\min_x \|Ax - b\|_2$, to computing the **minimum-norm** solution of sparse, underdetermined problems and to the solution of sparse **symmetric positive definite** linear systems. It can obviously be used for solving unsymmetric square problems in which case the stability provided by the use of orthogonal transformations comes at the cost of a higher operation count with respect to solvers based on, e.g., the LU factorization such as MUMPS. `qr_mumps` supports **real and complex**, **single or double** precision arithmetic.

As in all the sparse, direct solvers, the solution is achieved in three distinct phases:

Analysis in this phase an analysis of the structural properties of the input matrix is performed in preparation for the numerical factorization phase. This includes computing a column permutation which reduces the amount of *fill-in* coefficients (i.e., nonzeroes introduced by the factorization). This step does not perform any floating-point operation and is, thus, commonly much faster than the factorization and solve (depending on the number of right-hand sides) phases.

Factorization at this step, the actual QR or Cholesky factorization is computed. This step is the most computationally intense and, therefore, the most time consuming.

Solution once the factorization is done, the factors can be used to compute the solution of the problem through two operations:

Solve this operation computes the solution of the triangular system $Rx = b$ or $R^T x = b$;

Apply this operation applies the Q orthogonal matrix to a vector, i.e., $y = Qx$ or $y = Q^T x$.

These three steps have to be done in order but each of them can be performed multiple times. If, for example, the problem has to be solved against multiple right-hand sides (not all available at once), the analysis and factorization can be done only once while the solution is repeated for each right-hand side. By the same token, if the coefficients of a matrix are updated but not its structure, the analysis can be performed only once for multiple factorization and solution steps.

`qr_mumps` is based on the multifrontal factorization method. This method was first introduced by Duff and Reid [10] as a method for the factorization of sparse, symmetric linear systems and, since then, has been the object of numerous studies and the method of choice for several, high-performance, software packages such as MUMPS [4] and UMFPACK [8]. The method used in `qr_mumps` is described in full details in [7, 2].

`qr_mumps` is built upon the large knowledge base and know-how developed by the members of the MUMPS project. However, `qr_mumps` does not share any code with the MUMPS package and it is a

completely independent software. qr_mumps is developed and maintained in a collaborative effort by the APO team at the IRIT laboratory in Toulouse and the LaBRI laboratory in Bordeaux, France.

2 Features

2.1 Types of problems

qr_mumps can handle unsymmetric and symmetric, positive definite problems. In the first case it will use a QR factorization whereas, in the second, it will use a Cholesky factorization. In order to choose one or the other method, qr_mumps must be informed about the type of the problem through the sym field of the zqrm_spmat_type data structure: 0 means that the problem is unsymmetric and ¿0 means symmetric, positive definite. Note that in the second case, only half of the matrix must be provided, i.e., if the coefficient (i, j) is provided (j, i) must not be given.

2.2 Multithreading

qr_mumps is a parallel, multithreaded software based on the StarPU runtime system [5] and it currently supports multicore or, more generally, shared memory multiprocessor computers. qr_mumps does not run on distributed memory (e.g. clusters) parallel computers. Parallelism is achieved through a decomposition of the workload into fine-grained computational tasks which basically correspond to the execution of a BLAS or LAPACK operation on a blocks. It is strongly recommended to use sequential BLAS and LAPACK libraries and let qr_mumps have full control of the parallelism.

The number of threads/cores used by qr_mumps can be controlled in two different ways:

- by setting QRM_NCPU environment variable to the desired number of threads. In this case the number of threads will be the same throughout the execution of your program/application;
- through the qrm_init. This method has higher priority than the QRM_NCPU environment variable.

The granularity of the tasks is controlled by the qrm_mb_ and qrm_nb_ parameters (see Section qrm_set) which set the block size for partitioning internal data. Smaller values mean more parallelism; however, because this blocking factor is an upper-bound for the granularity of operations (or, more precisely for the granularity of calls to BLAS and LAPACK routines), it is recommended to choose reasonably large values in order to achieve high efficiency.

2.3 GPU acceleration

qr_mumps can leverage the computing power of Nvidia GPU, commonly available on modern supercomputing systems, to accelerate the solution of linear systems, especially large size ones. The use of GPUs is achieved through the StarPU runtime [1].

The number of GPUs used by qr_mumps can be controlled in two different ways:

- by setting QRM_NGPU environment variable to the desired number of GPUs. In this case the number of GPUs will be the same throughout the execution of your program/application;
- through the qrm_init. This method has higher priority than the QRM_NGPU environment variable.

  Note that it is possible to use multiple streams per GPU; this can be controlled through the StarPU STARPU_NWORKER_PER_CUDA environment variable.

2.4 Fill-reducing permutations

qr_mumps supports multiple methods for reducing the factorization fill-in through matrix column permutations. The choice is controlled through the qrm_ordering_ control parameter. Nested-dissection based methods are available through the packages Metis [13] and SCOTCH [14] packages as well as average minimum degree through the COLAMD [9] one. Nested-dissection based methods usually lead

to lower fill-in which ultimately results in faster and less memory consuming factorization. COLAMD, instead, typically leads to a faster execution of the analysis phase although is not as effective in reducing the fill-in which may result in a slower and more memory consuming factorization. Because the overall execution time is commonly dominated by the factorization, nested-dissection methods are usually more effective especially for large size problems. `qr_mumps` also allows the user to provide their own permutation, as explained in Secion [[*Control parameters.

2.5 Memory consumption control

`qr_mumps` allows for controlling the amount of memory used in the parallel factorization stage. In the multifrontal method, the memory consumption varies greatly throughout the sequential factorization reaching a maximum value which is referred to as the sequential peak (sp). Parallelism can considerably increase this peak because, in order to feed the working threads, more data is allocated at the same time which results in higher concurrency. In `qr_mumps` it is possible to bound the memory consumption of the factorization phase through the `qrm_mem_relax` parameter. If this parameter is set to a real value $x \geq 1$, the memory consumption will be bounded by $x \times sp$. Clearly, the tighter is this upper bound, the slower the factorization will proceed. Note that sp only includes the memory consumed by the factorization operation; moreover, although in practice it is possible to precisely pre-compute this value in the analysis phase, this may be expensive and thus `qrm_analyse` only computes a (hopefully) slight overestimation. The value of sp is available upon completion of the analysis phase through the `qrm_e_facto_mempeak` information parameter (see Section 5).

2.6 Asynchronous interface

An asynchronous interface is provided for the analysis, factorization apply and solve operations, respectively `qrm_analyse_async`, `qrm_factorize_async`, `qrm_apply_async` and `qrm_solve_async`. These routines are non-blocking variants of the `zqrm_analyse`, `zqrm_factorize`, `zqrm_apply` and `zqrm_solve`; this means that they will submit to the StarPU runtime system all the tasks the corresponding operating is composed of and will return control to the calling program as soon as possible. The completion of the tasks will be achieved asynchronously and can only be ensured through a call to the `qrm_barrier` routine. This has a number of advantages; for example it allows for executing concurrently operations that work on different data (e.g. the factorization of different matrices) or to pipeline the execution of operations which work on the same data (for example factorization and solve with the same matrix), in which case StarPU will take care of ensuring that the precedence constraints between tasks are respected. The asynchronous routine take an additional argument `qrm_dscr` which is a communication descriptor, i.e. a container for the submitted tasks; this has to be initialized through the `qrm_dscr_init` routine and destroyed using the `qrm_dscr_destroy` one.

Two main differences exist with respect to the synchronous interface:

- Right-hand sides must be registered to `qr_mumps` by means of the `zqrm_rhs_init` routine which associates a rank-1 or rank-2 Fortran array to a `zqrm_rhs_type` data structure.
- a communication descriptor must be initialized and passed to the operation routines: all the associated tasks will be submitted to this descriptor.

The completion of the operation can be guaranteed by calling a `qrm_barrier` routine either with the (optional) descriptor argument, in which case the routine will wait for all the tasks in that descriptor, or without, in which case the routine will wait for all the previously submitted tasks in all descriptors.

There is currently no support for asynchronous execution in the `qr_mumps` C interface.

See Section 2.6 for an example.

3 API

qr_mumps is developed in the Fortran 2008 language but includes a portable C interface developed through the Fortran iso_c_binding feature. Most of the qr_mumps features are available from both interfaces although the Fortran one takes full advantage of the language features, such as the interface overloading, that are not available in C. The naming convention used in qr_mumps groups all the routine or data type names into two families depending on whether they depend on the arithmetic or not. Typed names always begin by _qrm_ where the first underscore becomes d, s, z, c for real double, real single, complex double or complex single arithmetic, respectively. Untyped names, instead, simply begin by qrm_. Note that thanks to interface overloading in Fortran all the typed interfaces of a routine can be conveniently grouped into a single untyped one; this is described in details in Section 3.4. All the interfaces described in the remainder of this section are for the complex, double precision case. The interfaces for complex single, real double and real single can be obtained by replacing zqrm with cqrm, dqrm and sqrm, respectively and complex(real64) with complex(real32), real(real64), real(real32), respectively. Note that real64 and real32 are defined in the iso_fortran_env Fortran 2008 module and correspond to kind(1.d0) and kind(1.e0) or 8 and 4, respectively on basically all common compilers/architectures. All the routines that take vectors as input (e.g., zqrm_apply) can be called with either one vector (i.e. a rank-1 Fortran array x(:)) or multiple ones (i.e., a rank-2 Fortran array x(:,:)) through the same interface thanks to interface overloading. This is not possible for the C interface, in which case an extra argument is present in order to specify the number of vectors which are expected to be stored in column-major (i.e., Fortran style) format.

In this section only the Fortran API is presented. For each Fortran name (either of a routine or of a data type) the corresponding C name is obtained by adding the _c suffix. The number, type and order of arguments in the C routines is the same except for those routines that take dense vectors in which case, the C interface needs an extra argument specifying the number of vectors passed trough the same pointer. The user can refer to the code examples and to the zqrm_mumps.h file for the full details of the C interface.

3.1 Data types

3.1.1 zqrm_spmat_type

This data type is used to store a sparse matrix in the COO (or coordinate) format through the irn, jcn and val fields containing the row indices, column indices and values, respectively and the m, n and nz containing the number of rows, columns and nonzeroes, respectively. qr_mumps uses a Fortran-style 1-based numbering and thus all row indices are expected to be between 1 and m and all the column indices between 1 and n. Duplicate entries are summed during the factorization, out-of-bound entries are ignored. The sym field is used to specify if the matrix is symmetric (> 0) or unsymmetric ($= 0$).

```
type zqrm_spmat_type
   integer                    :: m
   integer                    :: n
   integer                    :: nz
   integer                    :: sym
   integer, pointer           :: irn(:)
   integer, pointer           :: jcn(:)
   complex(real64), pointer   :: val(:)
end type zqrm_spmat_type
```

3.1.2 zqrm_spfct_type

This type is used to set the parameters that control the behavior of a sparse factorization, to collect information about its execution (number of flops, memory consumpnion etc) and store the result of

the factorization, namely, the factors with all the symbolic information needed to use them in the solve phase.

```
type zqrm_spfct_type
   integer                  :: icntl(:)
   real(real32)             :: rcntl(:)
   integer(int64)           :: gstats(:)
   integer, pointer         :: cperm_in(:)
end type zqrm_spfct_type
```

- `cperm_in`: this array can be used to provide a matrix column permutation and is only accessed by `qr_mumps` in this case.
- `icntl`: this array contains all the integer control parameters. Its content can be modified either directly or indirectly through the `qrm_set` routine (see Section 3.3.3).
- `gstats`: this array contains all the statistics collected by `qr_mumps`. Its content can be accessed either directly or indirectly through the `qrm_get` routine (see Section 3.3.4).

3.2 Computational routines

3.2.1 zqrm_analyse

This routine performs the analysis phase (see Section 1) on A or A^T.

```
interface qrm_analyse
   subroutine zqrm_analyse(qrm_spmat, qrm_spfct, transp, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     type(zqrm_spfct_type)    :: qrm_spfct
     character, optional      :: transp
     integer, optional        :: info
   end subroutine zqrm_analyse
end interface qrm_analyse
```

Arguments:

- `qrm_spmat`: the input matrix of `zqrm_spmat_type`.
- `qrm_spfct`: the sparse factorization object of `zqrm_spfct_type`.
- `transp`: whether the input matrix should be transposed or not. Can be either `'t'` (`'c'` in in complex arithmetic) or `'n'`. In the Fortran interface this parameter is optional and set by default to `'n'` if not passed.
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.2 zqrm_factorize

This routine performs the factorization phase (see Section 1) on A or A^T. It can only be executed if the analysis is already done.

```
interface qrm_factorize
   subroutine zqrm_factorize(qrm_spmat, qrm_spfct, transp, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     type(zqrm_spfct_type)    :: qrm_spfct
     character, optional      :: transp
     integer, optional        :: info
   end subroutine zqrm_factorize
end interface qrm_factorize
```

Arguments:

- `qrm_spmat`: the input matrix of `zqrm_spmat_type`.
- `qrm_spfct`: the sparse factorization object of `zqrm_spfct_type`.
- `transp`: whether the input matrix should be transposed or not. Can be either `'t'` (`'c'` in in complex arithmetic) or `'n'`. In the Fortran interface this parameter is optional and set by default to `'n'` if not passed.
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.3 zqrm_apply

This routine computes $b = Q \cdot b$ or $b = Q^T \cdot b$. It can only be executed once the factorization is done.

```
interface zqrm_apply
   subroutine zqrm_apply2d(qrm_spfct, transp, b, info)
     type(zqrm_spfct_type)    :: qrm_spfct
     complex(real64)          :: b(:,:)
     character(len=*)         :: transp
     integer, optional        :: info
   end subroutine zqrm_apply2d
   subroutine zqrm_apply1d(qrm_spfct, transp, b, info)
     type(zqrm_spfct_type)    :: qrm_spfct
     complex(real64)          :: b(:)
     character(len=*)         :: transp
     integer, optional        :: info
   end subroutine zqrm_apply1d
end interface zqrm_apply
```

Arguments:

- `qrm_spfct`: the sparse factorization object resulting from `zqrm_factorize`
- `transp`: whether to apply Q or Q^T. Can be either `'t'` (`'c'` in complex arithmetic) or `'n'`.
- `b`: the b vector(s) to which Q or Q^T is applied.
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.4 zqrm_solve

This routine solves the triangular system $R \cdot x = b$ or $R^T \cdot x = b$. It can only be executed once the factorization is done.

```
interface zqrm_solve
   subroutine zqrm_solve2d(qrm_spfct, transp, b, x, info)
     type(zqrm_spfct_type)    :: qrm_spfct
     complex(real64)          :: b(:,:)
     complex(real64)          :: x(:,:)
     character(len=*)         :: transp
     integer, optional        :: info
   end subroutine zqrm_solve2d
   subroutine zqrm_solve1d(qrm_spfct, transp, b, x, info)
     type(zqrm_spfct_type)    :: qrm_spfct
     complex(real64)          :: b(:)
     complex(real64)          :: x(:)
     character(len=*)         :: transp
     integer, optional        :: info
   end subroutine zqrm_solve1d
end interface zqrm_solve
```

Arguments:

- `qrm_spfct`: the sparse factorization object resulting from `zqrm_factorize`
- `transp`: whether to solve for R or R^T. Can be either `'t'` (`'c'` in complex arithmetic) or `'n'`.
- `b`: the b right-hand side(s).
- `x`: the x solution vector(s).
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.5 zqrm_least_squares

This subroutine can be used to solve a linear least squares problem $\min_x \|Ax - b\|_2$ in the case where the input matrix is square or overdetermined. It is a shortcut for the sequence

```
call zqrm_analyse(qrm_spmat, qrm_spfct, 'n', info)
call zqrm_factorize(qrm_spmat, qrm_spfct, 'n', info)
call zqrm_apply(qrm_spfct, 'c', b, info)
call zqrm_solve(qrm_spfct, 'n', b, x, info)
```

Note that A^T can be used instead of A, in which case A^T must be overdetermined.

```
interface qrm_least_squares
   subroutine zqrm_least_squares2d(qrm_spmat, b, x, transp, cperm, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64)          :: b(:,:), x(:,:)
     character, optional      :: transp
     integer, optional        :: cperm(:)
     integer, optional        :: info
   end subroutine zqrm_least_squares2d
   subroutine zqrm_least_squares1d(qrm_spmat, b, x, transp, cperm, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64)          :: b(:), x(:)
     character, optional      :: transp
     integer, optional        :: cperm(:)
     integer, optional        :: info
   end subroutine zqrm_least_squares1d
end interface qrm_least_squares
```

Arguments:

- `qrm_spmat`: the input matrix.
- `b`: the b right-hand side(s). Will be modified.
- `x`: the x solution vector(s).
- `transp`: whether to use A or A^T.
- `cperm`: an optional integer array to pass a fill-reducing matrix column permutation.
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.6 zqrm_min_norm

This subroutine can be used to solve a linear minimum norm problem in the case where the input matrix is square or underdetermined. It is a shortcut for the sequence

```
call qrm_analyse(qrm_spmat, 'c', info)
call qrm_factorize(qrm_spmat, 'c', info)
call qrm_solve(qrm_spmat, 'c', b, x, info)
call qrm_apply(qrm_spmat, 'n', x, info)
```

Note that A^T can be used instead of A, in which case A^T must be underdetermined.

```
interface qrm_min_norm
   subroutine zqrm_min_norm2d(qrm_spmat, b, x, transp, cperm, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64)          :: b(:,:), x(:,:)
     character, optional      :: transp
     integer, optional        :: cperm(:)
     integer, optional        :: info
   end subroutine zqrm_min_norm2d
   subroutine zqrm_min_norm1d(qrm_spmat, b, x, transp, cperm, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64)          :: b(:), x(:)
     character, optional      :: transp
     integer, optional        :: cperm(:)
     integer, optional        :: info
   end subroutine zqrm_min_norm1d
end interface qrm_min_norm
```

Arguments:

- `qrm_spmat`: the input matrix.
- `b`: the b right-hand side(s).
- `x`: the x solution vector(s).
- `transp`: whether to use A or A^T
- `cperm`: an optional integer array to pass a fill-reducing matrix row permutation (i.e., a column permutation for A^T).
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.7 `zqrm_spposv`

This subroutine can be used to solve a linear symmetric, positive definite problem. It is a shortcut for the sequence

```
x = b
call qrm_analyse(qrm_spmat, 'n', info)
call qrm_factorize(qrm_spmat, 'n', info)
call qrm_solve(qrm_spmat, 'c', x, b, info)
call qrm_solve(qrm_spmat, 'n', b, x, info)
```

```
interface qrm_min_norm
   subroutine zqrm_min_norm2d(qrm_spmat, b, x, cperm, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64)          :: b(:,:), x(:,:)
     integer, optional        :: cperm(:)
     integer, optional        :: info
   end subroutine zqrm_min_norm2d
   subroutine zqrm_min_norm1d(qrm_spmat, b, x, cperm, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64)          :: b(:), x(:)
     integer, optional        :: cperm(:)
     integer, optional        :: info
   end subroutine zqrm_min_norm1d
end interface qrm_min_norm
```

Arguments:

- `qrm_spmat`: the input matrix.
- `b`: the b right-hand side(s). Will be modified.
- `x`: the x solution vector(s).
- `cperm`: an optional integer array to pass a fill-reducing matrix row permutation (i.e., a column permutation for A^T).
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.8 `zqrm_spmat_mv`

This subroutine performs a matrix-vector product of the type $y = \alpha Ax + \beta y$ or $y = \alpha A^T x + \beta y$.

```
interface zqrm_spmat_mv
   subroutine zqrm_spmat_mv_2d(qrm_spmat, transp, alpha, x, beta, y)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64),         :: y(:,:)
     complex(real64),         :: x(:,:)
     complex(real64),         :: alpha, beta
     character(len=*)         :: transp
   end subroutine zqrm_spmat_mv_2d
   subroutine zqrm_spmat_mv_1d(qrm_spmat, transp, alpha, x, beta, y)
     type(zqrm_spmat_type)    :: qrm_spmat
     complex(real64),         :: y(:)
     complex(real64),         :: x(:)
     complex(real64),         :: alpha, beta
     character(len=*)         :: transp
   end subroutine zqrm_spmat_mv_1d
end interface zqrm_spmat_mv
```

Arguments:

- `qrm_spmat`: the input matrix.
- `transp`: whether to multiply by A or A^T. Can be either `'t'` (`'c'` if in complex arithmetic) or `'n'`.
- `alpha`, `beta` the α and β scalars
- `y`: the y vector(s).
- `x`: the x vector(s).

3.2.9 `qrm_spmat_nrm`

This routine computes the one $\|A\|_1$ or the infinity $\|A\|_\infty$ or the Frobenius $\|A\|_F$ norm of a matrix.

```
interface qrm_spmat_nrm
   subroutine zqrm_spmat_nrm(qrm_spmat, ntype, nrm, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     real(real64)             :: nrm
     character                :: ntype
     integer, optional        :: info
   end subroutine zqrm_spmat_nrm
end interface qrm_spmat_nrm
```

Arguments:

- `qrm_spmat`: the input matrix.

- `ntype`: the type of norm to be computed. It can be either `'i'`, `'1'` or `'f'` for the infinity, one and Frobenius norms, respectively.
- `nrm`: the computed norm.
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.10 `zqrm_vecnrm`

This routine computes the one-norm $\|x\|_1$, the infinity-norm $\|x\|_\infty$ or the two-norm $\|x\|_2$ of a vector.

```
interface qrm_vecnrm
   subroutine zqrm_vecnrm2d(vec, n, ntype, nrm, info)
     complex(real64)          :: vec(:,:)
     real(real64)             :: nrm(:)
     integer                  :: n
     character                :: ntype
     integer, optional        :: info
   end subroutine zqrm_vecnrm2d
   subroutine zqrm_vecnrm1d(vec, n, ntype, nrm, info)
     complex(real64)          :: vec(:)
     real(real64)             :: nrm
     integer                  :: n
     character                :: ntype
     integer, optional        :: info
   end subroutine zqrm_vecnrm1d
end interface qrm_vecnrm
```

Arguments:

- `x`: the x vector(s).
- `n`: the size of the vector.
- `ntype`: the type of norm to be computed. It can be either `'i'`, `'1'` or `'2'` for the infinity, one and two norms, respectively.
- `nrm` the computed norm(s). If `x` is a rank-2 array (i.e., a multivector) this argument has to be a rank-1 array `nrm(:)` and each of its elements will contain the norm of the corresponding column of `x`.
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.11 `qrm_residual_norm`

This routine computes the scaled norm of the residual $\frac{\|b-Ax\|_\infty}{\|b\|_\infty+\|x\|_\infty\|A\|_\infty}$, i.e., the normwise backward error. It is a shortcut for the sequence

```
call qrm_vecnrm(b, qrm_spmat%m, 'i', nrmb)
call qrm_vecnrm(x, qrm_spmat%n, 'i', nrmx)
call qrm_spmat_mv(qrm_spmat, 'n', -1, x, 1, b)
call qrm_spmat_nrm(qrm_spmat, 'i', nrma)
call qrm_vecnrm(b, qrm_spmat%m, 'i', nrmr)
nrm = nrmr/(nrmb+nrma*nrmx)
```

Note that A^T can be used instead of A.

```
interface qrm_residual_norm
   subroutine zqrm_residual_norm2d(qrm_spmat, b, x, nrm, transp, info)
     real(real64)             :: nrm(:)
     type(zqrm_spmat_type)    :: qrm_spmat
```

```
    complex(real64)          :: b(:,:), x(:,:)
    character, optional      :: transp
    integer, optional        :: info
  end subroutine zqrm_residual_norm2d
  subroutine zqrm_residual_norm1d(qrm_spmat, b, x, nrm, transp, info)
    real(real64)             :: nrm
    type(zqrm_spmat_type)    :: qrm_spmat
    complex(real64)          :: b(:), x(:)
    character, optional      :: transp
    integer, optional        :: info
  end subroutine zqrm_residual_norm1d
end interface qrm_residual_norm
```

Arguments:

- `qrm_spmat`: the input matrix.
- `b`: the b right-hand side(s). On output this argument contains the residual.
- `x`: the x solution vector(s).
- `nrm` the scaled residual norm. This argument is of type `real` for single precision arithmetic (both real and complex) and `real(kind(1.d0))` for double precision ones (both real and complex). If `x` and `b` are rank-2 arrays (i.e., multivectors) this argument has to be a rank-1 array `nrm(:)` and each coefficient will contain the scaled norm of the residual for the corresponding column of `x` and `b`.
- `transp`: whether to use A or A^T
- `info`: an optional output parameter that returns the exit status of the routine.

3.2.12 `qrm_residual_orth`

Computes the quantity $\frac{\|A^T r\|_2}{\|r\|_2}$ which can be used to evaluate the quality of the solution of a least squares problem (see [6], page 34). It is a shortcut for the sequence

```
call qrm_spmat_mv(qrm_spmat, 'c', 1, r, 0, atr)
call qrm_vecnrm(r, qrm_spmat%m, '2', nrmr)
call qrm_vecnrm(atr, qrm_spmat%n, '2',  nrm)
nrm = nrm/nrmr
```

Note that A^T can be used instead of A.

```
interface qrm_residual_orth
  subroutine zqrm_residual_orth2d(qrm_spmat, r, nrm, transp, info)
    real(real64)             :: nrm(:)
    type(zqrm_spmat_type)    :: qrm_spmat
    complex(real64)          :: r(:,:)
    character, optional      :: transp
    integer, optional        :: info
  end subroutine zqrm_residual_orth2d
  subroutine zqrm_residual_orth1d(qrm_spmat, r, nrm, transp, info)
    real(real64)             :: nrm
    type(zqrm_spmat_type)    :: qrm_spmat
    complex(real64)          :: r(:)
    character, optional      :: transp
    integer, optional        :: info
  end subroutine zqrm_residual_orth1d
end interface qrm_residual_orth
```

Arguments:

- `qrm_spmat`: the input problem.
- `r`: the r residual(s).
- `nrm` the scaled $A^T r$ norm. This argument is of type `real(real64)` for double precision arithmetic (both real and complex) and `real(real32)` for single precision ones (both real and complex). If `r` is a rank-2 array (i.e., a multivector) this argument has to be a rank-1 array `nrm(:)` and each coefficient will contain the scaled norm of $A^T r$ for the corresponding column of `r`.
- `transp`: whether to use A or A^T
- `info`: an optional output parameter that returns the exit status of the routine.

3.3 Management routines

3.3.1 qrm_init

This routine initializes `qr_mumps` and should be called prior to any other `qr_mumps` routine.

```
subroutine qrm_init(ncpu, ngpu, info)
    integer, optional          :: ncpu, ngpu, info
end subroutine qrm_init
```

Arguments:

- `ncpu`: an optional input parameter that sets the number of working threads. If not specified, the `QRM_NCPU` is used.
- `ngpu`: an optional input parameter that sets the number of working threads. If not specified, the `QRM_NGPU` is used.
- `info`: an optional output parameter that returns the exit status of the routine.

3.3.2 qrm_finalize

This routine finalizes `qr_mumps` and no other `qr_mumps` routine should be called afterwards.

```
subroutine qrm_finalize()
end subroutine qrm_finalize
```

3.3.3 qrm_set

This family of routines is used to set control parameters that define the behavior of `qr_mumps`; it is possible to set default parameters which are applied to all the following factorizations or the parameters of a specific sparse factorization object of type `zqrm_spfct_type`. In the Fortran API the `qrm_set` interfaces overloads all of them. These control parameters are explained in full details in Section 4.

```
interface qrm_set
   subroutine qrm_glob_set_i4(string, ival, info)
     character(len=*)          :: string
     integer                   :: ival
     integer, optional         :: info
   end subroutine qrm_glob_set_i4
   subroutine qrm_glob_set_r4(string, rval, info)
     character(len=*)          :: string
     real(real32)              :: rval
     integer, optional         :: info
   end subroutine qrm_glob_set_r4
   subroutine zqrm_spfct_set_i4(qrm_spfct, string, ival, info)
     type(zqrm_spfct_type)     :: qrm_spfct
```

```
      character(len=*)          :: string
      integer                   :: ival
      integer, optional         :: info
    end subroutine zqrm_spfct_set_i4
    subroutine zqrm_spfct_set_r4(qrm_spfct, string, rval, info)
     type(zqrm_spfct_type)      :: qrm_spfct
     character(len=*)           :: string
     real(real32)               :: rval
     integer, optional          :: info
     subroutine zqrm_spfct_set_r4
end interface qrm_set
```

Arguments:

- `qrm_fct`: the sparse factorization object.
- `string`: a string describing the parameter to be set (see Section 4 for a full list).
- `val`: the parameter value.
- `info`: an optional output parameter that returns the exit status of the routine.

3.3.4 `qrm_get`

This family of routines can be used to get the value of a control parameter or the get the value of information collected by `qr_mumps` during the execution (see Section 4 for a full list).

```
interface qrm_get
    subroutine qrm_glob_get_i4(string, ival, info)
      character(len=*)          :: string
      integer                   :: ival
      integer, optional         :: info
    end subroutine qrm_glob_get_i4
    subroutine qrm_glob_get_i8(string, iival, info)
      character(len=*)          :: string
      integer(int64)            :: iival
      integer, optional         :: info
    end subroutine qrm_glob_get_i8
    subroutine qrm_glob_get_r4(string, rval, info)
      character(len=*)          :: string
      real(real32)              :: rval
      integer, optional         :: info
    end subroutine qrm_glob_get_r4
    subroutine zqrm_spfct_get_i4(qrm_spfct, string, ival, info)
      type(zqrm_spfct_type)     :: qrm_spfct
      character(len=*)          :: string
      integer                   :: ival
      integer, optional         :: info
    end subroutine zqrm_spfct_get_i4
    subroutine zqrm_spfct_get_i8(qrm_spfct, string, ival, info)
      type(zqrm_spfct_type)     :: qrm_spfct
      character(len=*  )        :: string
      integer(int64)            :: ival
      integer, optional         :: info
    end subroutine zqrm_spfct_get_i8
end interface qrm_get
```

Arguments:

- `qrm_spfct`: the sparse factorization object.

- `string`: a string describing the parameter to be set (see Sections 4 and 5 for a full list).
- `val`: the returned parameter value.
- `info`: an optional output parameter that returns the exit status of the routine.

3.3.5 `zqrm_spmat_init`

This routine initializes a `zqrm_spmat_type` data structure. This amounts to nullifying all the pointers and setting the rest of the data fileds to 0.

```
interface qrm_spmat_init
   subroutine zqrm_spmat_init(qrm_spmat, info)
     type(zqrm_spmat_type)    :: qrm_spmat
     integer, optional        :: info
   end subroutine zqrm_spmat_init
end interface qrm_spmat_init
```

Arguments:

- `qrm_spmat`: the matrix to be initialized.
- `info`: an optional output parameter that returns the exit status of the routine.

3.3.6 `zqrm_spfct_init`

This routine initializes a `zqrm_spfct_type` data structure. This amounts to setting all the control parameters to the default values.

```
interface qrm_spfct_init
   subroutine zqrm_spfct_init(qrm_spfct, qrm_spmat, info)
     type(zqrm_spfct_type)    :: qrm_spfct
     type(zqrm_spmat_type)    :: qrm_spmat
     integer, optional        :: info
   end subroutine zqrm_spfct_init
end interface qrm_spfct_init
```

Arguments:

- `qrm_spfct`: the sparse factorization object to be initialized.
- `info`: an optional output parameter that returns the exit status of the routine.

3.3.7 `zqrm_spfct_destroy`

This routine cleans up a `zqrm_spfct_type` data structure by deleting the result of a sparse factorization.

```
interface qrm_spfct_destroy
   subroutine zqrm_spfct_destroy(qrm_spfct, info)
     type(zqrm_spfct_type)    :: qrm_spfct
     integer, optional        :: info
   end subroutine zqrm_spfct_destroy
end interface qrm_spfct_destroy
```

Arguments:

- `qrm_spfct`: the sparse factorization object to be destroyed.
- `info`: an optional output parameter that returns the exit status of the routine.

3.3.8 `qrm_alloc` and `qrm_dealloc`

These routines are used to allocate and deallocate Fortran `pointers` or `allocatables`. They're essentially wrappers around the Fortran `allocate` function and they're mostly used internally by `qr_mumps` too keep track of the amount of memory allocated. Input pointers and allocatables can be either 1D or 2D, integer, real or complex, single precision or double precision (all of these are available regardless of the arithmetic with which `qr_mumps` has been compiled). For the sake of brevity, only the interface of the 1D and 2D, double precision, complex versions is given below.

```
interface qrm_alloc
   subroutine qrm_aalloc_z(a, m, info)
     complex(real64), allocatable  :: a(:)
     integer                       :: m
     integer, optional             :: info
   end subroutine qrm_aalloc_z
   subroutine qrm_aalloc_2z(a, m, n, info)
     complex(real64), allocatable  :: a(:,:)
     integer                       :: m, n
     integer, optional             :: info
   end subroutine qrm_aalloc_2z
   subroutine qrm_palloc_z(a, m, info)
     complex(real64), pointer      :: a(:)
     integer                       :: m
     integer, optional             :: info
   end subroutine qrm_palloc_z
   subroutine qrm_palloc_2z(a, m, n, info)
     complex(real64), pointer      :: a(:,:)
     integer                       :: m, n
     integer, optional             :: info
   end subroutine qrm_palloc_2z
end interface qrm_alloc

interface qrm_dealloc
   subroutine qrm_adealloc_z(a, info)
     complex(real64), allocatable  :: a(:)
     integer, optional             :: info
   end subroutine qrm_adealloc_z
   subroutine qrm_adealloc_2z(a, info)
     complex(real64), allocatable  :: a(:,:)
     integer, optional             :: info
   end subroutine qrm_adealloc_2z
   subroutine qrm_pdealloc_z(a, info)
     complex(real64), pointer      :: a(:)
     integer, optional             :: info
   end subroutine qrm_pdealloc_z
   subroutine qrm_pdealloc_2z(a, info)
     complex(real64), pointer      :: a(:,:)
     integer, optional             :: info
   end subroutine qrm_pdealloc_2z
end interface qrm_dealloc
```

Arguments:

- `a`: the input 1D or 2D pointer or allocatable array.
- `m`: the row size.
- `n`: the column size.
- `info`: an optional output parameter that returns the exit status of the routine.

3.4 Interface overloading

The interface overloading feature of the Fortran language is heavily used inside `qr_mumps`. First of all, all the typed routines of the type `_qrm_xyz` are overloaded with a generic `qrm_xyz` interface. This means that, for example, a call to the `qrm_factorize` routine will result in a call to `sqrm_factorize` or as a call to `dqrm_factorize` depending on whether the input matrix is of type `sqrm_spmat_type` or `dqrm_spmat_type`, respectively (i.e., single or double precision real, respectively). As said in Sections 3.3.3 and 3.3.4, the `qrm_set` and `qrm_get` interfaces overload the routines in the corresponding families and the same holds for the allocation/deallocation routines (see Section 3.3.8). The advantages of the overloading are obvious. Take the following example:

```
type(sqrm_spmat_type)     :: qrm_spmat
real(real32), allocatable :: b(:), x(:)

! initialize the control data structure.
call qrm_spmat_init(qrm_spmat)
...
! allocate arrays for the input matrix
call qrm_alloc(qrm_spmat%irn, nz)
call qrm_alloc(qrm_spmat%jcn, nz)
call qrm_alloc(qrm_spmat%val, nz)
call qrm_alloc(b, m)
call qrm_alloc(x, n)

! initialize the data
...

! solve the problem
call qrm_least_squares(qrm_spmat, b, x)
...
```

In case the user wants to switch to double precision, only the declarations on the first two lines have to be modified and the rest of the code stays unchanged.

4 Control parameters

Control parameters define the behavior of `qr_mumps` and can be set in two modes:

- global mode: in this mode it possible to either set generic parameters (e.g., the unit for output or error messages) or default parameter values (e.g., the ordering method to be used on the problem) that apply to all `zqrm_spfct_type` objects that are successively initialized through the `zqrm_spfct_init` routine.
- problem mode: these parameters control the behavior of `qr_mumps` on a specific sparse factorization problem. Because the `zqrm_spfct_init` routine sets the control parameters to their default values, these have to be modified after the sparse factorization object initialization.

All the control parameters can be set through the `qrm_set` routine (see the interface in Section 3.3); problem specific control parameters can also be set by manually changing the coefficients of the `qrm_spfct_type%icntl` array (note the underscore in this case); alternatively, the default values of all parameters can be set through the corresponding environment variables:

```
type(zqrm_spmat_type) :: qrm_spmat
type(zqrm_spfct_type) :: qrm_spfct1, qrm_spfct2
```

```
! set default ordering method to scotch
call qrm_set('qrm_ordering', qrm_scotch_)

call qrm_spfct_init(qrm_spfct1, qrm_spmat)
call qrm_spfct_init(qrm_spfct2, qrm_spmat)

! set the block size to 256 only for qrm_spfct2
call qrm_set(qrm_spfct2, 'qrm_mb', 256)

! set the block size to 128 only for qrm_spfct1
qrm_spfct1%icntl(qrm_mb_) = 128

...
```

Here is a list of the parameters, their meaning and the accepted values; for each parameter `omp_param` a corresponding `OMP_PARAM` environment variable exists which can be used to set its default value.

- `qrm_ncpu` (global, int32): `val` is an integer specifying the number of CPU cores to use for the subsequent `qr_mumps` calls. This has to be set prior to the call to `qrm_init`. This value can also be set either through the `QRM_NCPU` environment variable (lowest priority) or passed directly as an argument to the `qrm_init` routine (highest priority). Default is 1. This is a global parameter and cannot be set for a specific problem only.
- `qrm_ngpu` (global, int32): `val` is an integer specifying the number of GPUs to use for the subsequent `qr_mumps` calls. This has to be set prior to the call to `qrm_init`. This value can also be set either through the `QRM_NGPU` environment variable (lowest priority) or passed directly as an argument to the `qrm_init` routine (highest priority. Default is 0. This is a global parameter and cannot be set for a specific problem only.
- `qrm_ounit` (global, int32): `val` is an integer specifying the unit for output messages; if negative, output messages are suppressed. Default is 6. This is a global parameter and cannot be set for a specific problem only.
- `qrm_eunit` (global, int32): `val` is an integer specifying the unit for error messages; if negative, error messages are suppressed. Default is 0. This is a global parameter and cannot be set for a specific problem only.
- `qrm_ordering` (both, int32): this parameter specifies what permutation to apply to the columns of the input matrix in order to reduce the fill-in and, consequently, the operation count of the factorization and solve phases. This parameter is used by `qr_mumps` during the analysis phase and, therefore, has to be set before it starts. The following pre-defined values are accepted:
  - `qrm_auto_`: the choice is automatically made by `qr_mumps`. This is the default.
  - `qrm_natural_`: no permutation is applied.
  - `qrm_given_`: a column permutation is provided by the user through the `qrm_spmat_type%cperm_in`.
  - `qrm_colamd_`: the COLAMD software package (if installed) is used for computing the column permutation.
  - `qrm_scotch_`: the SCOTCH software package (if installed) is used for computing the column permutation.
  - `qrm_metis_`: the Metis software package (if installed) is used for computing the column permutation.
- `qrm_keeph` (both, int32): this parameter says whether the H matrix should be kept for later use or discarded. This parameter is used by `qr_mumps` during the factorization phase and, therefore, has to be set before it starts. Accepted value are:

  - qrm_yes_: the H matrix is kept. This is the default.
  - qrm_no_: the H matrix is discarded.

- qrm_mb and qrm_nb (both, int32): These parameters define the block-size (rows and columns, respectively) for data partitioning and, thus, granularity of parallel tasks. Smaller values mean higher concurrence. This parameter, however, implicitly defines an upper bound for the granularity of call to BLAS and LAPACK routines (defined by the qrm_ib parameter described below); therefore, excessively small values may result in poor performance. This parameter is used by qr_mumps during the analysis and factorization phases and, therefore, has to be set before these start. The default value is 256 for both. Note that qrm_mb has to be a multiple of qrm_nb.
- qrm_ib (both, int32): this parameter defines the granularity of BLAS/LAPACK operations. Larger values mean better efficiency but imply more fill-in and thus more flops and memory consumption (please refer to [2] for more details). The value of this parameter is upper-bounded by the qrm_nb parameter described above. This parameter is used by qr_mumps during the factorization phase and, therefore, has to be set before it starts. The default value is 32. It is strongly advised to choose, for this parameter, a submultiple of qrm_nb.
- qrm_bh (both, int32): this parameter defines the type of algorithm for the communication-avoiding QR factorization of frontal matrices (see the details in [2]. Smaller values mean more concurrency but worse tasks efficiency; if lower or equal to zero the largest possible value is chosen for each front. Default value is -1.
- qrm_rhsnb (both, int32): in the case where multiple right-hand sides are passed to the qrm_apply or the qrm_solve routines, this parameter can be used to define a blocking of the right-hand sides. This parameter is used by qr_mumps during the solve phase and, therefore, has to be set before it starts. By default, all the right-hand sides are treated in a single block.
- qrm_pinth (both, int32): an integer value to control memory pinning when GPUs are used: all frontal matrices whose size (min(rows,cols)) is bigger than this value will be pinned.
- qrm_mem_relax (both, real32): a real(real32) value (>= 1) that sets a relaxation parameter, with respect to the sequential peak, for the memory consumption in the factorization phase. If negative, the memory consumption is not bounded. Default value is -1.0. See Section 2.5 for the details of this feature.
- qrm_rd_eps (both, int32): a real(real32) value setting a threshold to estimate the rank of the problem. If > 0 the zqrm_factorize routine will count the number of diagonal coefficients of the R factor whose absolute value is smaller than the provided value. This number can be retrieved through the qrm_rd_num information parameter described in the next section.

5 Information parameters

Information parameters return information about the behavior of qr_mumps and can be either global or problem specific.

All the information parameters can be gotten through the qrm_get routine (see the interface in Section 3.3.4); problem specific control parameters can also be retrieved by manually reading the coefficients of the qrm_spfct_type%gstats array.

The qrm_get routine can also be used to retrieve the values of all the control parameters described in the previous section with the obvious usage.

- qrm_max_mem (global, int64): this parameter, of type integer(int64) returns the maximum amount of memory allocated by qr_mumps during its execution.
- qrm_tot_mem (global, int64): this parameter, of type integer(int64) returns the total amount of memory allocated by qr_mumps at the moment when the qrm_get routine is called.

- `qrm_e_nnz_r` (local, int64): this parameter, of type `integer(int64)` returns an estimate, computed during the analysis phase, of the number of nonzero coefficients in the R factor. This value is only available after the `qrm_analyse` routine is executed.
- `qrm_e_nnz_h` (local, int64): this parameter, of type `integer(int64)` returns an estimate, computed during the analysis phase, of the number of nonzero coefficients in the H matrix. This value is only available after the `qrm_analyse` routine is executed.
- `qrm_e_facto_flops` (local, int64): this parameter, of type `integer(int64)` returns an estimate, computed during the analysis phase, of the number of floating point operations performed during the factorization phase. This value is only available after the `qrm_analyse` routine is executed.
- `qrm_nnz_r` (local, int64): this parameter, of type `integer(int64)` returns the actual number of the nonzero coefficients in the R factor after the factorization is done. This value is only available after the `qrm_factorize` routine is executed.
- `qrm_nnz_h` (local, int64): this parameter, of type `integer(int64)` returns the actual number of the nonzero coefficients in the H matrix after the factorization is done. This value is only available after the `qrm_factorize` routine is executed.
- `qrm_e_facto_mempeak` (local, int64): this parameter, of type `integer(int64)` returns an estimate of the peak memory consumption of the factorization operation.
- `qrm_rd_num` (local, int32): this information parameter returns the number of diagonal coefficients of the R factor whose absolute value is lower than `qrm_rd_eps` if this control parameter was set to a value greater than 0.

6 Error handling

Most `qr_mumps` routines have an optional argument `info` (which is always last) that returns the exit status. If the routine succeeded `info` will be equal to 0 otherwise it will have a positive value. A message will be printed on the `qrm_eunit` unit (see Section 4 upon occurrence of an error. A list of error codes:

- **1**: The provided sparse matrix format is not supported.
- **3**: `qrm_spfct%cntl` is invalid.
- **4**: Trying to allocate an already allocated allocatable or pointer.
- **5-6**: Memory allocation problem.
- **8**: Input column permutation not provided or invalid.
- **9**: The requested ordering method is unknown.
- **10**: Internal error: insufficient size for array .
- **11**: Internal error: Error in lapack routine.
- **12**: Internal error: out of memory.
- **13**: The analysis must be done before the factorization.
- **14**: The factorization must be done before the solve.
- **15**: This type of norm is not implemented.
- **16**: Requested ordering method not available (i.e., has not been installed).
- **17**: Internal error: error from call to subroutine...

- **18**: An error has occured in a call to COLAMD.
- **19**: An error has occured in a call to SCOTCH.
- **20**: An error has occured in a call to Metis.
- **23**: Incorrect argument to `qrm_set=/=qrm_get`.
- **25**: Internal error: problem opening file.
- **27**: Incompatible values in `qrm_spfct%icntl`.
- **28**: Incorrect value for `qrm_mb_=/=qrm_nb_=/=qrm_ib_`.
- **29**: Incorrect value for `qrm_spmat%m=/=n=/=nz`.
- **30**: `qrm_apply` cannot be called if the H matrix is discarded.
- **31**: StarPU initialization error.
- **32**: Matrix is rank deficient.

7 Examples

7.1 Standard interface

The code below shows a basic example program that allocates and fills up a sparse matrix, runs the analysis, factorization and solve on it, computes the solution backward error and finally prints some information collected during the process.

```
program zqrm_example

  use zqrm_mod
  implicit none

  type(zqrm_spmat_type)          :: qrm_spmat
  complex(r64), allocatable      :: b(:), x(:), r(:), xe(:)
  integer                        :: info
  real(r64)                      :: anrm, bnrm, xnrm, rnrm, onrm, fnrm

  call qrm_init()

  ! initialize the matrix data structure.
  call qrm_spmat_init(qrm_spmat)
  call qrm_spmat_alloc(qrm_spmat, 13, 7, 5, 'coo')

  qrm_spmat%irn = (/1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7/)
  qrm_spmat%jcn = (/1, 3, 5, 2, 3, 5, 1, 4, 4, 5, 2, 1, 3/)
  qrm_spmat%val = (/1.d0, 2.d0, 3.d0, 1.d0, 1.d0, 2.d0, 4.d0,
                  & 1.d0, 5.d0, 1.d0, 3.d0, 6.d0, 1.d0/)

  call qrm_alloc(b,  qrm_spmat%m, info)
  call qrm_alloc(r,  qrm_spmat%m, info)
  call qrm_alloc(x,  qrm_spmat%n, info)
  call qrm_alloc(xe, qrm_spmat%n, info)
  b = (/22.d0, 5.d0, 13.d0, 8.d0, 25.d0, 5.d0, 9.d0/)
  xe = (/1.d0, 2.d0, 3.d0, 4.d0, 5.d0/)

  r = b
  call qrm_vecnrm(b, size(b,1), '2', bnrm)

  call qrm_least_squares(qrm_spmat, b, x)

  call qrm_residual_norm(qrm_spmat, r, x, rnrm)
  call qrm_vecnrm(x, qrm_spmat%n, '2', xnrm)
  call qrm_spmat_nrm(qrm_spmat, 'f', anrm)
  call qrm_residual_orth(qrm_spmat, r, onrm)

  write(*,'("Expected result is x= 1.00000 2.00000 3.00000 4.00000 5.00000")
   ')
  write(*,'("Computed result is x=",5(x,f7.5))')x

  xe = xe-x;
  call qrm_vecnrm(xe, qrm_spmat%n, '2', fnrm)
  write(*,'(" ")')
  write(*,'("Forward error norm       ||xe-x||  = ",e7.2)')fnrm
  write(*,'("Optimality residual norm ||A^T*r|| = ",e7.2)')onrm

  call qrm_spmat_destroy(qrm_spmat)
  call qrm_dealloc(b)
  call qrm_dealloc(r)
```

```
  call qrm_dealloc(x)

  stop
end program zqrm_example
```

7.2 Asynchronous interface

```
program zqrm_example

  use zqrm_mod
  implicit none


  type(zqrm_spmat_type)          :: qrm_spmat
  type(zqrm_spfct)               :: qrm_spfct
  type(qrm_dscr)                 :: qrm_dscr
  type(zqrm_sdata_type)          :: x_rhs, b_rhs
  complex(r64), allocatable      :: b(:), x(:), r(:), xe(:)
  integer                        :: info
  real(r64)                      :: anrm, bnrm, xnrm, rnrm, onrm, fnrm

  call qrm_init()

  ! initialize the matrix data structure.
  call qrm_spmat_init(qrm_spmat)

  call qrm_spmat_alloc(qrm_spmat, 13, 7, 5, 'coo')

  qrm_spmat%irn = (/1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7/)
  qrm_spmat%jcn = (/1, 3, 5, 2, 3, 5, 1, 4, 4, 5, 2, 1, 3/)
  qrm_spmat%val = (/1.d0, 2.d0, 3.d0, 1.d0, 1.d0, 2.d0, 4.d0,
                 & 1.d0, 5.d0, 1.d0, 3.d0, 6.d0, 1.d0/)

  call qrm_alloc(b,  qrm_spmat%m, info)
  call qrm_alloc(r,  qrm_spmat%m, info)
  call qrm_alloc(x,  qrm_spmat%n, info)
  call qrm_alloc(xe, qrm_spmat%n, info)
  b = (/22.d0, 5.d0, 13.d0, 8.d0, 25.d0, 5.d0, 9.d0/)
  xe = (/1.d0, 2.d0, 3.d0, 4.d0, 5.d0/)

  r = b
  call qrm_vecnrm(b, size(b,1), '2', bnrm)

  ! init the sparse fato object
  call zqrm_spfct_init(qrm_spfct, qrm_spmat, err)

  ! init the descriptor
  call qrm_dscr_init(qrm_dscr)

  ! init the rhs and solution data
  call zqrm_sdata_init(b_rhs, b)
  call zqrm_sdata_init(x_rhs, x)

  call zqrm_analyse_async(qrm_dscr, qrm_mat, qrm_spfct)
  call zqrm_factorize_async(qrm_dscr, qrm_mat, qrm_spfct)
  call qrm_apply_async(qrm_dscr, qrm_spfct, qrm_conj_transp, b_rhs)
  call qrm_solve_async(qrm_dscr, qrm_spfct, qrm_no_transp, b_rhs, x_rhs)
```

```
  call qrm_barrier()

  call qrm_residual_norm(qrm_spmat, r, x, rnrm)
  call qrm_vecnrm(x, qrm_spmat%n, '2', xnrm)
  call qrm_spmat_nrm(qrm_spmat, 'f', anrm)
  call qrm_residual_orth(qrm_spmat, r, onrm)

  write(*,'("Expected result is x= 1.00000 2.00000 3.00000 4.00000 5.00000")
   ')
  write(*,'("Computed result is x=",5(x,f7.5))')x

  xe = xe-x;
  call qrm_vecnrm(xe, qrm_spmat%n, '2', fnrm)
  write(*,'(" ")')
  write(*,'("Forward error norm       ||xe-x||  = ",e7.2)')fnrm
  write(*,'("Optimality residual norm ||A^T*r|| = ",e7.2)')onrm

  call qrm_spmat_destroy(qrm_spmat)
  call qrm_spfct_destroy(qrm_spfct)
  call qrm_dscr_destroy(qrm_dscr)
  call qrm_sdata_destroy(b_rhs)
  call qrm_sdata_destroy(x_rhs)
  call qrm_dealloc(b)
  call qrm_dealloc(r)
  call qrm_dealloc(x)


  stop
end program zqrm_example
```

7.3 C interface

```
int main(){
  struct zqrm_spmat_type_c qrm_spmat;
  int i;
  double rnrm, onrm, anrm, bnrm, xnrm;
  int irn[13] = {1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7};
  int jcn[13] = {1, 3, 5, 2, 3, 5, 1, 4, 4, 5, 2, 1, 3};
  double _Complex val[13] = {1.0, 2.0, 3.0, 1.0, 1.0,
                             2.0, 4.0, 1.0, 5.0, 1.0,
                             3.0, 6.0, 1.0};
  double _Complex b[7] = {22.0, 5.0, 13.0, 8.0, 25.0, 5.0, 9.0};
  double _Complex r[7] = {22.0, 5.0, 13.0, 8.0, 25.0, 5.0, 9.0};
  double _Complex xe[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  double _Complex x[5];

  qrm_init_c(-1, -1);

  /* initialize the matrix data structure */
  zqrm_spmat_init_c(&qrm_spmat);
  qrm_spmat.m   = 7;
  qrm_spmat.n   = 5;
  qrm_spmat.nz  = 13;
  qrm_spmat.irn = irn;
  qrm_spmat.jcn = jcn;
  qrm_spmat.val = val;
```

```
  zqrm_least_squares_c(&qrm_spmat, b, x, 1);

  zqrm_residual_norm_c(&qrm_spmat, r, x, 1, &rnrm);
  zqrm_residual_orth_c(&qrm_spmat, r, 1, &onrm);
  zqrm_vecnrm_c(x, qrm_spmat.n, 1, '2', &xnrm);
  zqrm_vecnrm_c(b, qrm_spmat.m, 1, '2', &bnrm);
  zqrm_spmat_nrm_c(&qrm_spmat, 'f', &anrm);

  printf("Expected result is x= 1.00000 2.00000 3.00000 4.00000 5.00000\n");
  printf("Computed result is x= ");
  for(i=0; i<5; i++){
    printf("%7.5f ",creal(x[i]));
    x[i] -= xe[i];
  }
  printf("\n");
  zqrm_vecnrm_c(x, qrm_spmat.n, 1, '2', &xnrm);
  printf("Forward error            ||xe-x||  = %10.5e\n",xnrm);
  printf("Optimality residual norm ||A^T*r|| = %10.5e\n",onrm);

  zqrm_spmat_destroy_c(&qrm_spmat);

  qrm_finalize_c();
  return 0;

}
```

8 Performance tuning

The performance of `qr_mumps` depends on a number of parameters. Default values are provided for these parameters that are expected to achieve reasonably good performance on a wide range of problems and architectures but for optimal performance these should be tuned. In this section we provide a list of these parameters and explain how do they have an effect on performance.

Block size `qr_mumps` decomposes frontal matrices into blocks of size $mb \times mb$ (set through the `qrm_mb` control parameter described in section 4); this decomposition provides an additional level of parallelism (other than that already expressed by the elimination tree) because it is possible to execute concurrently tasks that operate on different blocks. On the one hand, small values of mb provide high parallelism; on the other hand, high values of mb provide high efficiency for each task and make the tasks scheduling overhead negligible. This parameter should be, therefore, chosen as to provide the best compromise between parallelism and tasks efficiency. The optimal value depends on the size and structure of the problem, the number and features of processing units, the efficiency and scalability of BLAS operations etc. On current CPUs block sizes of 128 or 256 achieve close to optimal task performance and good parallelism on moderately sized problems; is GPUs are used, higher block sizes (1024) provide better performance. Choosing a large mb value to achieve high performance on GPU devices can severely reduce parallelism and lead to CPU starvation. In this case the nb parameter (`qrm_nb`) can be used to generate additional parallelism; if this parameter is set to a submultiple of mb, the dynamic, hierarchical partitioning technique described in [1] is used which can lead to better performance. Finally, some tasks use an internal block size; this is set by the ib parameter (`qrm_ib` which has to be a submultiple of mb and nb) and defines a compromise between efficiency of tasks and overall amount of floating point operations. Again, when GPUs are used, larger values of ib lead to better speed whereas on CPUs values of 32/64 provide satisfactory speed.

Reduction tree shape the *bh* parameter (`qrm_bh`) defines the shape of the reduction tree in the *QR* panel reduction as explained in [12] or [2] . A value of k means that a panel is divided in groups of size k, intra-group reduction is done with a flat tree, inter-group reduction with a binary tree. Therefore, a value of one achieves the highest parallelism because the whole panel is reduced through a binary tree. Conversely a value which is equal or higher than the number of blocks in a panel leads to lower parallelism because all the blocks in the panels are reduced one after the other; a zero or negative value sets a flat tree on all panels in all fronts of the multifrontal factorization. Nevertheless it must be noted that excessively small values of *bh* may lead to inefficient computations because of the nature of the involved tasks. A flat tree typically achieves high performance on a wide range of problems but for very overdetermined problems it may be beneficial to use hybrid trees.

Ordering fill-reducing ordering is essential to limit the fill-in produced by the factorization. This ordering (set through the `qrm_ordering` control parameter) is computed during the analysis phase and corresponds to a matrix permutation that defines the order in which unknowns are eliminated. The ordering will also affect the shape of the elimination tree which can be more or less balanced or deep with obvious consequences on parallelism, efficiency and, ultimately, execution time. Nested Dissection [11] methods, such as those implemented in the Metis and SCOTCH packages, usually provide the best results and their running time may be high; local orderings such as AMD/COLAMD [3] typically have a lower running time, which results in a faster analysis step, but lead to higher fill-in and thus higher running time and memory consumption for the factorization and the solve.

GPU streams when GPUs are used, it can be helpful (and it usually is) to use multiple streams per GPU to allow a single GPU to execute multiple tasks concurrently. Using multiple GPU streams is especially beneficial to achieve high GPU occupancy when a relatively small block size *mb* is chosen to prevent CPU starvation. This can be controlled through the `STARPU_NWORKER_PER_CUDA` StarPU environment variable. By default one stream is active per GPU device and higher performance can be commonly achieved with values of 2 up to 20.

9 Credits

- The following people have contributed to the development of `qr_mumps` code: Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Florent Lopez, Ian Masliah, Antoine Jego.
- The design of `qr_mumps` is deeply influenced by the countless advices from Patrick Amestoy, Jean-Yves L'Excellent and Chiara Puglisi, the developers of the MUMPS and HSL MA49 solvers.
- We are also grateful to the StarPU development team for their constant support in the use of this runtime system.
- Thanks Arnaud Legrand, Lucas Schnorr and Luka Stanisic for their feedback on performance analysis through SimGrid and StarVZ.
- `qr_mumps` was partially funded by the ANR SOLHAR and SOLHARIS projects.
- The development, testing and performance evaluation of `qr_mumps` are possible thanks to the computing resources provided by the CALMIP and PlaFRIM supercomputing centers as well as those of the GENCI consortium.

10 Bibliography

References

[1] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Task-based multifrontal QR solver for GPU-accelerated multicore architectures. In *HiPC*, pages 54–63. IEEE Computer Society, 2015. **Best paper award**.

[2] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Softw.*, 43(2):13:1–13:22, August 2016.

[3] Patrick R. Amestoy, T. A. Davis, and Iain S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 33(3):381–388, 2004.

[4] Patrick R. Amestoy, Iain S. Duff, J. Koster, and Jean-Yves L'Excellent. MUMPS: a general purpose distributed memory sparse solver. In A. H. Gebremedhin, F. Manne, R. Moe, and T. Sørevik, editors, *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21*, pages 122–131. Springer-Verlag, 2000. Lecture Notes in Computer Science 1947.

[5] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[6] Å. Björck. *Numerical methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

[7] Alfredo Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. *SIAM Journal on Scientific Computing*, 35(4):C323–C345, 2013.

[8] T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

[9] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30:353–376, September 2004.

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

[11] Alan J. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10(2):345–363, 1973.

[12] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Tile QR factorization with parallel panel processing for multicore architectures. In *IPDPS*, pages 1–10. IEEE, 2010.

[13] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[14] F. Pellegrini. Scotch 5.0 User's guide. Technical Report, LaBRI, Université Bordeaux I, August 2007.