qr_mumps

Version 2.0

Users guide

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 factorization of the input matrix. Therefore, it is suited to solving sparse **least-squares** problems $\min_x \|Ax - b\|_2$ and to computing the **minimum-norm** solution of sparse, underdetermined problems. It can obviously be used for solving 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. 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 mush faster than the factorization and solve (depending on the number of right-hand sides) phases.

Factorization : at this step, the actual QR 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 built upon the large knowledge base and know-how developed by the members of the MUMPS[1] 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 Algorithm

qr_mumps is based on the multifrontal factorization method. This method was first introduced by Duff and Reid [7] 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 [2] and UMFPACK [6]. At the heart of this method is the concept of an *elimination tree*, extensively studied and formalized later by Liu [8]. This tree graph describes the dependencies among computational tasks in the multifrontal factorization. The multifrontal method can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix $A^T A$ share the same structure under the hypothesis that the matrix A is *Strong Hall* (for a definition of this property see, for example, [4]). Based on this equivalence, the elimination tree for the QR factorization of A is the same as that for the Cholesky factorization of $A^T A$. In the case where the Strong Hall property does not hold, the elimination tree related to the Cholesky factorization of $A^T A$ can still be used although the resulting QR factorization will perform more computations and consume more memory than what is really

[1] http://mumps.enseeiht.fr

needed; alternatively, the matrix A can be permuted to a Block Triangular Form (BTF) where all the diagonal blocks are Strong Hall.

In a basic multifrontal method, the elimination tree has n nodes, where n is the number of columns in the input matrix A, each node representing one pivotal step of the QR factorization of A. Every node of the tree is associated with a dense matrix, known as *frontal matrix* that contains all the coefficients affected by the elimination of the corresponding pivot. The whole QR factorization consists in a bottom-up traversal of the tree where, at each node, two operations are performed:

- **assembly**: a set of rows from the original matrix is assembled together with data produced by the processing of child nodes to form the frontal matrix;
- **factorization**: one Householder reflector is computed and applied to the whole frontal matrix in order to annihilate all the subdiagonal elements in the first column. This step produces one row of the R factor of the original matrix and a complement which corresponds to the data that will be later assembled into the parent node (commonly referred to as a *contribution block*). The Q factor is defined implicitly by means of the Householder vectors computed on each front; the matrix that stores the coefficients of the computed Householder vectors, will be referred to as the H matrix from now on.

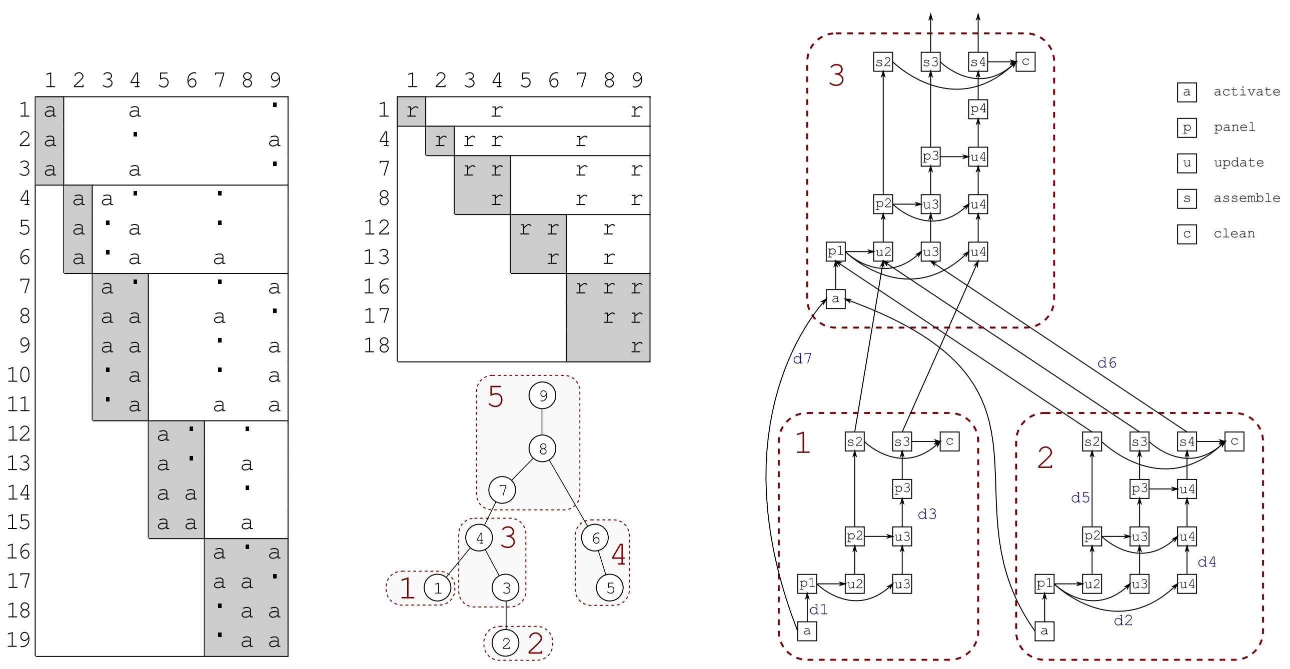

Figure 1: *(left)* Example of multifrontal QR factorization. The dots denote the fill-in coefficients. *(right)* The DAG associated with supernodes 1, 2 and 3 for a block-column size of one.

In practical implementations of the multifrontal QR factorization, and in `qr_mumps` , nodes of the elimination tree are amalgamated to form *super nodes*. The amalgamated pivots correspond to rows of R that have the same structure and can be eliminated at once within the same frontal matrix without producing any additional fill-in in the R factor. The elimination of amalgamated pivots and the consequent update of the trailing frontal submatrix can thus be performed by means of efficient Level-3 BLAS routines. Moreover, amalgamation reduces the number of assembly operations increasing the computations-to-communications ratio which results in better performance. The amalgamated elimination tree is also commonly referred to as *assembly tree*. Figure 1 *(left)* shows a sparse matrix along with the assembly tree and the resulting R factor.

Parallelism is exploited, in `qr_mumps` , through a fine-grained decomposition of the frontal matrices into blocks. As illustrated in Figure 1 *(right)*, this allows for representing the whole matrix factorization as DAG (Directed Acyclic Graph) where nodes represent sequential tasks, i.e. the execution of one elementary operation on a block-column, and edges the dependencies among them.

The tasks in the DAG are then scheduled dynamically according to a data-flow parallel execution model. This approach delivers high flexibility and concurrence which result in high performance on modern, multicore computers.

The method used in `qr_mumps` is described in full details in [5, 1].

3 Features

3.1 Parallelism

`qr_mumps` is a parallel, multithreaded software based on the StarPU runtime system [3] 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. As described in Section 2, 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 used by `qr_mumps` can be controlled in two different ways:

1. by setting `QRM_NUM_THREADS` 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;
2. through the `qrm_init` (see Section 4.3.1. This method has higher priority than the `QRM_NUM_THREADS` environment variable.

The granularity of the tasks is controlled by the `qrm_mb_` and `qrm_nb_` parameters (see Section 6.2) 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.

3.2 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 $x.y \geq 1$, the memory consumption will be bounded by $x.y \times sp$. 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 7.2.

4 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 4.4. All the interfaces described in the remainder of this section are for the real, single precision case. The interfaces for real double, complex single and complex double can be obtained by replacing sqrm with dqrm, cqrm and zqrm, respectively and real with real(kind(1.d0)), complex, complex(kind(1.d0)), respectively. All the routines that take vectors as input (e.g., _qrm_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 _qrm_mumps.h file for the full details of the C interface.

4.1 Data types

4.1.1 _qrm_spmat_type

This data type is used to define a problem and all the information needed to process it. Specifically it contains the problem matrix, the parameters used to control the behavior of the qr_mumps operations done on it and the statistics collected by qr_mumps during the execution of these operations.

```
type sqrm_spmat_type
   ! Row and column indices
   integer, pointer :: irn(:), jcn(:)
   ! Numerical values
   real, pointer :: val(:)
   ! Number of rows, columns
   ! and nonzeroes
   integer :: m, n, nz
   ! A pointer to an array
   ! containing a column permutation
   ! provided by the user
   integer, pointer :: cperm_in(:)
   ! Integer control parameters
   integer :: icntl(20)
   ! Collected statistics
   integer(kind=8) :: gstats(10)
end type sqrm_spmat_type
```

- Matrix data: matrices can be stored 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.
- `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 6.2).
- `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 7.2).

4.2 Computational routines

4.2.1 `qrm_analyse`

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

```
interface qrm_analyse

   subroutine sqrm_analyse(qrm_mat, transp, info)
     type(sqrm_spmat_type):: qrm_mat
     character, optional  :: transp
     integer, optional    :: info
   end subroutine sqrm_analyse

end interface qrm_analyse
```

Arguments:

- `qrm_mat`: the input problem.
- `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.

4.2.2 `qrm_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 sqrm_factorize(qrm_mat, transp, info)
     type(sqrm_spmat_type):: qrm_mat
     character, optional  :: transp
     integer, optional    :: info
   end subroutine sqrm_factorize

end interface qrm_factorize
```

Arguments:

- `qrm_mat`: the input problem.
- `transp`: whether the input matrix should be transposed or not. Can be either `'t'` (`'c'` 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.

4.2.3 `qrm_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 qrm_apply

   subroutine sqrm_apply1d(qrm_mat, transp, b, info)
     type(sqrm_spmat_type) :: qrm_mat
     character             :: transp
     real                  :: b(:)
     integer, optional     :: info
   end subroutine sqrm_apply1d

   subroutine sqrm_apply2d(qrm_mat, transp, b, info)
     type(sqrm_spmat_type) :: qrm_mat
     character             :: transp
     real                  :: b(:,:)
     integer, optional     :: info
   end subroutine sqrm_apply2d

end interface qrm_apply
```

Arguments:

- `qrm_mat`: the input problem.
- `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.

4.2.4 `qrm_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 qrm_solve

   subroutine sqrm_solve1d(qrm_mat, transp, b, x, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:)
     real                  :: x(:)
     character             :: transp
     integer, optional     :: info
   end subroutine sqrm_solve1d

   subroutine sqrm_solve2d(qrm_mat, transp, b, x, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:,:)
     real                  :: x(:,:)
     character             :: transp
     integer, optional     :: info
   end subroutine sqrm_solve2d

end interface qrm_solve
```

Arguments:

- `qrm_mat`: the input problem.
- `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.

4.2.5 `qrm_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 qrm_analyse(qrm_mat, 'n', info)
call qrm_factorize(qrm_mat, 'n', info)
call qrm_apply(qrm_mat, 't', b, info)
call qrm_solve(qrm_mat, 'n', b, x, info)
```

```
interface qrm_least_squares

   subroutine sqrm_least_squares1d(qrm_mat, b, x, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:)
     real                  :: x(:)
     integer, optional     :: info
   end subroutine sqrm_least_squares1d

   subroutine sqrm_least_squares2d(qrm_mat, b, x, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:,:)
     real                  :: x(:,:)
     integer, optional     :: info
   end subroutine sqrm_least_squares2d

end interface qrm_least_squares
```

Arguments:

- `qrm_mat`: the input problem.

- 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.

4.2.6 qrm_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_mat, 't', info)
call qrm_factorize(qrm_mat, 't', info)
call qrm_solve(qrm_mat, 't', b, x, info)
call qrm_apply(qrm_mat, 'n', b, info)
```

```
interface qrm_min_norm
   subroutine sqrm_min_norm1d(qrm_mat, b, x, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: x(:)
     real                  :: b(:)
     integer, optional     :: info
   end subroutine sqrm_min_norm1d

   subroutine sqrm_min_norm2d(qrm_mat, b, x, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: x(:,:)
     real                  :: b(:,:)
     integer, optional     :: info
   end subroutine sqrm_min_norm2d

end interface qrm_min_norm
```

Arguments:

- qrm_mat: the input problem.
- 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.

4.2.7 qrm_matmul

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

```
interface qrm_matmul

   subroutine sqrm_matmul1d(qrm_mat, transp, alpha, x, beta, y)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: y(:)
     real                  :: x(:)
     real                  :: alpha, beta
     character             :: transp
   end subroutine sqrm_matmul1d

   subroutine sqrm_matmul2d(qrm_mat, transp, alpha, x, beta, y)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: y(:,:)
     real                  :: x(:,:)
     real                  :: alpha, beta
     character             :: transp
   end subroutine sqrm_matmul2d

end interface qrm_matmul
```

Arguments:

- `qrm_mat`: the input problem.
- `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
- `x`: the x vector(s).
- `y`: the y vector(s).

4.2.8 `qrm_matnrm`

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

```
interface qrm_matnrm

   subroutine sqrm_matnrm(qrm_mat, ntype, nrm, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: nrm
     character             :: ntype
     integer, optional     :: info
   end subroutine sqrm_matnrm

end interface qrm_matnrm
```

Arguments:

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

4.2.9 `qrm_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 sqrm_vecnrm1d(vec, n, ntype, nrm)
     real               :: vec(:)
     integer            :: n
     character          :: ntype
     real               :: nrm
     integer, optional :: info
   end subroutine sqrm_vecnrm1d

   subroutine sqrm_vecnrm2d(vec, n, ntype, nrm)
     real               :: vec(:,:)
     integer            :: n
     character          :: ntype
     real               :: nrm(:)
     integer, optional :: info
   end subroutine sqrm_vecnrm2d

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.

4.2.10 `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_mat%m, 'i', nrmb)
call qrm_vecnrm(x, qrm_mat%n, 'i', nrmx)
call qrm_matmul(qrm_mat, 'n', -1, x, 1, b)
call qrm_matnrm(qrm_mat, 'i', nrma)
call qrm_vecnrm(b, qrm_mat%m, 'i', nrmr)
nrm = nrmr/(nrmb+nrma*nrmx)
```

```
interface qrm_residual_norm

   subroutine sqrm_residual_norm1d(qrm_mat, b, x, nrm, info)
     type(sqrm_spmat_type) :: qrm_mat
     real                   :: b(:)
     real                   :: x(:)
     real                   :: nrm
```

```
      integer, optional      :: info
   end subroutine sqrm_residual_norm1d

   subroutine sqrm_residual_norm2d(qrm_mat, b, x, nrm, info)
      type(sqrm_spmat_type) :: qrm_mat
      real                  :: b(:,:)
      real                  :: x(:,:)
      real                  :: nrm
      integer, optional     :: info
   end subroutine sqrm_residual_norm2d

end interface qrm_residual_norm
```

Arguments:

- `qrm_mat`: the input problem.
- `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`.
- `info`: an optional output parameter that returns the exit status of the routine.

4.2.11 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 [4], page 34). It is a shortcut for the sequence

```
  call qrm_matmul(qrm_mat, 't', 1, r, 0, atr)
  call qrm_vecnrm(r,   qrm_mat%m, '2', nrmr)
  call qrm_vecnrm(atr, qrm_mat%n, '2',  nrm)
  nrm = nrm/nrmr
```

```
interface qrm_residual_orth

   subroutine sqrm_residual_orth1d(qrm_mat, r, nrm, info)
      type(sqrm_spmat_type) :: qrm_mat
      real                  :: r(:)
      real                  :: nrm
      integer, optional     :: info
   end subroutine sqrm_residual_orth1d

   subroutine sqrm_residual_orth2d(qrm_mat, r, nrm, info)
      type(sqrm_spmat_type) :: qrm_mat
      real                  :: r(:,:)
      real                  :: nrm
      integer, optional     :: info
   end subroutine sqrm_residual_orth2d

end interface qrm_residual_orth
```

Arguments:

- `qrm_mat`: the input problem.
- `r`: the r residual(s).
- `nrm` the scaled $A^T r$ 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 `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`.
- `info`: an optional output parameter that returns the exit status of the routine.

4.3 Management routines

4.3.1 `qrm_init`

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

```
  subroutine qrm_init(nthreads, info)
    integer, optional            :: nthreads
    integer, optional            :: info
  end subroutine qrm_init
```

Arguments:

- `nth`: an option input parameter that sets the number of working threads. If not specified, the `QRM_NUM_THREADS` is used.
- `info`: an optional output parameter that returns the exit status of the routine.

4.3.2 `qrm_finalize`

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

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

4.3.3 `qrm_set`

This family of routines is used to set control parameters that define the behavior of `qr_mumps` . In the Fortran API the `qrm_set` interfaces overloads all of them (see Section 4.4 for more details). These control parameters are explained in full details in Section 6.

```
interface qrm_set

   subroutine sqrm_pseti(qrm_mat, string, ival, info)
     type(sqrm_spmat_type) :: qrm_mat
     character(len=*)      :: string
     integer               :: ival
     integer, optional     :: info
   end subroutine sqrm_pseti
   subroutine qrm_gseti(string, ival, info)
     character(len=*)       :: string
     integer                :: ival
     integer, optional      :: info
   end subroutine qrm_gseti

end interface qrm_set
```

Arguments:

- qrm_mat: the input problem.
- string: a string describing the parameter to be set (see Section 6 for a full list).
- val: the parameter value.
- info: an optional output parameter that returns the exit status of the routine.

4.3.4 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 7 for a full list).

```
interface qrm_get

   subroutine sqrm_pgeti(qrm_mat, string, ival, info)
     type(sqrm_spmat_type) :: qrm_mat
     character(len=*)       :: string
     integer                :: ival
     integer, optional      :: info
   end subroutine sqrm_pgeti

   subroutine sqrm_pgetii(qrm_mat, string, ival, info)
     type(sqrm_spmat_type) :: qrm_mat
     character(len=*)       :: string
     integer(kind=8)        :: ival
     integer, optional      :: info
   end subroutine sqrm_pgetii

   subroutine qrm_ggeti(string, ival, info)
     character(len=*)       :: string
     integer                :: ival
     integer, optional      :: info
   end subroutine qrm_ggeti

   subroutine qrm_ggetii(string, ival, info)
     character(len=*)       :: string
     integer(kind=8)        :: ival
     integer, optional      :: info
   end subroutine qrm_ggetii

end interface qrm_get
```

Arguments:

- qrm_mat: the input problem.
- string: a string describing the parameter to be set (see Sections 6 and 7 for a full list).
- val: the returned parameter value.
- info: an optional output parameter that returns the exit status of the routine.

4.3.5 qrm_spmat_init

This routine initializes a qrm_spmat_type data structure. this pretty much amounts to setting default values for all the control parameters related to a specific problem. No other routine can be executed on qrm_spmat if it has not been initialized.

```
interface qrm_spmat_init

   subroutine sqrm_spmat_init(qrm_spmat, info)
     type(sqrm_spmat_type) :: qrm_mat
     integer, optional     :: info
   end subroutine sqrm_spmat_init

end interface qrm_spmat_init
```

Arguments:

- qrm_mat: the input problem.
- info: an optional output parameter that returns the exit status of the routine.

4.3.6 qrm_spmat_destroy

This routine destroys an instance of the qrm_spmat_type data structure. By default, this means that it will cleanup all the additional data that has been produced and attached to it (and that is not visible to the user) during the execution of the various qr_mumps operations. In the Fortran interface an optional all parameter is present which allows for deallocating also the arrays containing the input matrix, i.e., irn, jcn and val. Note that in this case the memory counters will be decremented (see Section 4.3.7) and thus it doesn't make much sense to pass all=.true. unless these arrays have been allocated through the qrm_alloc routine.

```
interface qrm_spmat_destroy

   subroutine sqrm_spmat_destroy(qrm_mat, all, info)
     type(sqrm_spmat_type) :: qrm_mat
     logical, optional     :: all
     integer, optional     :: info
   end subroutine sqrm_spmat_destroy

end interface qrm_spmat_destroy
```

Arguments:

- qrm_mat: the input problem.
- all: if set equal to .true. the original matrix arrays will be deallocated. This option is not available on the C interface.
- info: an optional output parameter that returns the exit status of the routine.

4.3.7 `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, single precision, real versions is given below.

```
interface qrm_alloc

   subroutine qrm_aalloc_s(a, m, info)
     real(kind(1.e0)), allocatable  :: a(:)
     integer                        :: m
     integer, optional              :: info
   end subroutine qrm_aalloc_s

   subroutine qrm_aalloc_2s(a, m, n, info)
     real(kind(1.e0)), allocatable  :: a(:,:)
     integer                        :: m, n
     integer, optional              :: info
   end subroutine qrm_aalloc_2s

   subroutine qrm_palloc_s(a, m, info)
     real(kind(1.e0)), pointer      :: a(:)
     integer                        :: m
     integer, optional              :: info
   end subroutine qrm_palloc_s

   subroutine qrm_palloc_2s(a, m, n, info)
     real(kind(1.e0)), pointer      :: a(:,:)
     integer                        :: m, n
     integer, optional              :: info
   end subroutine qrm_palloc_2s

end interface qrm_alloc


interface qrm_dealloc

   subroutine qrm_adealloc_s(a, info)
     real(kind(1.e0)), allocatable  :: a(:)
     integer, optional              :: info
   end subroutine qrm_adealloc_s

   subroutine qrm_adealloc_2s(a, info)
     real(kind(1.e0)), allocatable  :: a(:,:)
     integer, optional              :: info
   end subroutine qrm_adealloc_2s

   subroutine qrm_pdealloc_s(a, info)
     real(kind(1.e0)), pointer      :: a(:)
```

```
      integer, optional                :: info
    end subroutine qrm_pdealloc_s

    subroutine qrm_pdealloc_2s(a, info)
      real(kind(1.e0)), pointer        :: a(:,:)
      integer, optional                :: info
    end subroutine qrm_pdealloc_2s

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.

4.4 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(a)` routine will result in a call to `sqrm_factorize(a)` or as a call to `dqrm_factorize(a)` depending on whether `a` is of type `sqrm_spmat_type` or `dqrm_spmat_type`, respectively (i.e., single or double precision real, respectively). As said in Section 4.3 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 4.3.7. The advantages of the overloading are obvious. Take the following example:

```
  type(sqrm_spmat_type) :: qrm_mat
  real, allocatable     :: b(:), x(:)

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

  ! initialize the data
  ...

  ! solve the problem
  call qrm_least_squares(qrm_mat, 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.

5 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 7.1 upon occurrence of an error. A list of error codes:

1 : The provided sparse matrix format is not supported.

2 : Symmetric matrices are not supported.

3 : qrm_spmat%cntl is not associated or 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_spmat%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.

6 Control parameters

Control parameters define the behavior of `qr_mumps` and can be classified in two types:

- global: these parameters control the global behavior of `qr_mumps` and are not related to a specific problem, e.g., the unit for output messages.
- problem specific: these parameters control the behavior of `qr_mumps` on a specific problem, e.g., the ordering method to be used on the problem.

All the control parameters can be set through the `qrm_set` routine (see the interface in Section 4.3); problem specific control parameters can also be set by manually changing the coefficients of the `qrm_spmat_type%icntl` array.

6.1 Global parameters

The global parameters can be set through the `qrm_set` routine. A call

```
call qrm_set('qrm_param', val)
```

sets parameter `qrm_param` can be set to a value `val` where the latter can either be a preset value (a constant, predefined, value) or, simply, an integer.

Here is a list of the parameters, their meaning and the accepted values:

- `qrm_ounit`: `val` is an integer specifying the unit for output messages; if negative, output messages are suppressed. Default is 6.
- `qrm_eunit`: `val` is an integer specifying the unit for error messages; if negative, error messages are suppressed. Default is 0.

6.2 Problem specific parameters

The problem specific parameters can be set through the `qrm_set` routine. A call

```
call qrm_set(qrm_mat, 'qrm_param', val)
```

sets, for the `qrm_mat` problem, parameter `qrm_param` can be set to a value `val` where the latter can either be a preset value (a constant, predefined, value) or, simply, an integer. Equivalently, a problem specific control parameter can be set like (nothe the underscore at the end of `qrm_param_`:

```
qrm_mat%icntl(qrm_param_) = val
```

Here is a list of the parameters, their meaning and the accepted values:

- `qrm_ordering`: 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`: 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`: 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 values are 256 and 128, respectively. Note that `qrm_mb` has to be a multiple of `qrm_nb`.

- `qrm_ib`: 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 [5] 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`: this parameter defines the type of algorithm for the communication-avoiding QR factorization of frontal matrices (see the details in [1]). 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`: 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_mem_relax`: a real 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 3.2 for the details of this feature.

7 Information parameters

Information parameters return information about the behavior of `qr_mumps` and can be classified in two types:

- global: these parameters describe the global behavior of `qr_mumps` and are not related to a specific problem, e.g., the peak amount of memory consumed by `qr_mumps` .

- problem specific: these parameters describe the behavior of `qr_mumps` on a specific problem, e.g., the total number of flops executed during the factorization of a matrix.

All the information parameters can be gotten through the `qrm_get` routine (see the interface in Section 4.3); problem specific control parameters can also be retrieved by manually reading the coefficients of the `qrm_spmat_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.

7.1 Global parameters

```
call qrm_get('qrm_param', val)
```

- qrm_max_mem: this parameter, of type integer (or, better, of type integer(kind=8)), returns the maximum amount of memory allocated by qr_mumps during its execution.
- qrm_tot_mem: this parameter, of type integer (or, better, of type integer(kind=8)) , returns the total amount of memory allocated by qr_mumps at the moment when the qrm_get routine is called.

7.2 Problem specific parameters

```
call qrm_get(qrm_mat, 'qrm_param', val)
```

- qrm_e_nnz_r: this parameter, of type integer (or, better, of type integer(kind=8)) 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: this parameter, of type integer (or, better, of type integer(kind=8)) 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 this parameter, of type integer (or, better, of type integer(kind=8)) 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: this parameter, of type integer (or, better, of type integer(kind=8)) 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: this parameter, of type integer (or, better, of type integer(kind=8)) 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: this parameter, of type integer (or, better, of type integer(kind=8)) returns an estimate of the peak memory consumption of the factorization operation.

8 Example

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 sqrm_test_small
  use sqrm_mod
  implicit none

  type(sqrm_spmat_type)  :: qrm_mat
  integer                :: ierr, nargs, i, nrhs
  real, allocatable      :: b(:), x(:), r(:)
  real                   :: rnrm, onrm

  call qrm_init()

  ! initialize the control data structure.
  call qrm_spmat_init(qrm_mat)

  ! allocate arrays for the input matrix
  call qrm_alloc(qrm_mat%irn, 13)
  call qrm_alloc(qrm_mat%jcn, 13)
  call qrm_alloc(qrm_mat%val, 13)
  ! initialize the input matrix
  qrm_mat%jcn = (/1,1,1,2,2,3,3,3,3,4,4,5,5/)
  qrm_mat%irn = (/2,3,6,1,6,2,4,5,7,2,3,2,4/)
  qrm_mat%val = (/0.7,0.6,0.4,0.1,0.1,0.3,0.6,0.7,0.2,0.5,0.2,0.1,0.6/)
  qrm_mat%m   = 7
  qrm_mat%n   = 5
  qrm_mat%nz  = 13

  write(*,'("Starting␣Analysis")')
  call qrm_analyse(qrm_mat)

  write(*,'("Starting␣Factorization")')
  call qrm_factorize(qrm_mat)

  call qrm_alloc(b, qrm_mat%m)
  call qrm_alloc(r, qrm_mat%m)
  call qrm_alloc(x, qrm_mat%n)

  b = 1.e0
  ! as by is changed when applying Q', we save a copy in r for later use
  r = b
  call qrm_apply(qrm_mat, 't', b)
  call qrm_solve(qrm_mat, 'n', b, x)

  ! compute the residual
  call qrm_residual_norm(qrm_mat, r, x, rnrm)
  call qrm_residual_orth(qrm_mat, r, onrm)
  write(*,'("||r||/||A||␣␣␣␣=␣",e10.2)')rnrm
  write(*,'("||A^tr||/||r||␣=␣",e10.2)')onrm

  call qrm_dealloc(b)
  call qrm_dealloc(r)
  call qrm_dealloc(x)
  call qrm_spmat_destroy(qrm_mat, all=.true.)
  write(*,'("␣␣Nonzeroes␣in␣R␣␣␣␣␣␣␣␣␣␣␣:␣",i20)')qrm_mat%gstats(qrm_nnz_r_)
  write(*,'("␣␣Total␣flops␣at␣facto␣␣␣␣␣:␣",i20)')qrm_mat%gstats(qrm_e_facto_flops_)
  write(*,'("␣␣Global␣memory␣peak␣␣␣␣␣␣␣:␣",f9.3,"␣MB")') &
       &real(qrm_max_mem,kind(1.d0))/1024.d0/1024.d0

  call qrm_finalize()

  stop
end program sqrm_test_small
```

9 Asynchronous execution

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, i.e., each of them will submit to the StarPU runtime system all the tasks the corresponding operating is composed of and will return control to the calling program ass 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.

Here is an example of how these routines can be used:

```
type(sqrm_spmat_type)  :: qrm_mat
real(kind(1.e0))       :: b(:,:)
real(kind(1.e0))       :: x(:,:)

type(sqrm_rhs_type)    :: x_rhs, b_rhs
type(qrm_dscr_type)    :: qrm_dscr

call qrm_init()

!init the matrix data structure
call qrm_spmat_init(qrm_mat)

! fill up the matrix and b
! ...

! init the rhs data structures
call qrm_rhs_init(b_rhs, b)
call qrm_rhs_init(x_rhs, x)

! init the descriptor data structure
call qrm_dscr_init(qrm_dscr)

! submit analysis, facto, apply and solve operations
call qrm_analyse_async(qrm_dscr, qrm_mat, 'n')
call qrm_factorize_async(qrm_dscr, qrm_mat, 'n')

call qrm_apply_async(qrm_dscr, qrm_mat, 't', b_rhs, err)
call qrm_solve_async(qrm_dscr, qrm_mat, 'n', b_rhs, x_rhs, err)

! wait for their completion
call qrm_barrier(qrm_dscr)

! cleanup
call qrm_dscr_destroy(qrm_dscr)
call qrm_rhs_destroy(b_rhs)
call qrm_rhs_destroy(x_rhs)
call qrm_spmat_destroy(qrm_mat)
```

Two main differences exist with respect to the synchronous interface:

- Right-hand sides must be registered to `qr_mumps` by means of the `qrm_rhs_init` routine which associates a rank-1 or rank-2 Fortran array to a `sqrm_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.

Note that the code above corresponds (without RHS blocking) to the `qrm_least_squares` routine.

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

9.1 API

References

[1] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems. *ACM Transactions On Mathematical Software*, 2016. To appear.

[2] 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.

[3] 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.

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

[5] A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices, 2011. Submitted to SIAM SISC and APO technical report number RT-APO-11-6 [PDF].

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

[7] 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.

[8] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIMAX*, 11:134–172, 1990.