User's Guide for ParU, an unsymmetric multifrontal multithreaded sparse LU factorization package

Mohsen Aznaveh[*], Timothy A. Davis

VERSION 0.0.0, May 20, 2022

Abstract

ParU is an implementation of the multifrontal sparse LU factorization method. Parallelism is exploited both in the BLAS and across different frontal matrices using OpenMP tasking, a shared-memory programming model for modern multicore architectures. The package is written in C++ and real sparse matrices are supported.

1 Introduction

The algorithms used in ParU will be discussed in a companion paper, ?. This document gives detailed information on the installation and use of ParU. ParU is a parallel sparse direct solver. This package uses OpenMP tasking for parallelism. ParU calls UMFPACK for symbolic analysis phase, after that some symbolic analysis is done by ParU itself and then numeric phase starts. The numeric computation is a task parallel phase using OpenMP and each task calls parallel BLAS; i.e. nested parallism. The performance of BLAS has a heavy impact on the performance of ParU. However, depending on the input problem performance of parallelsim in BLAS sometimes does not have effects in ParU.

1.0.1 Instructions on using METIS

SuiteSparse now on METIS 5.1.0, which is distributed along with SuiteSparse itself. Its use is optional, however. ParU is using METIS as the default ordering. METIS tends to give orderings that are good for the parallelism. You can compile and run your code without using METIS; We recommend using METIS along with ParU.

Note that METIS is not bug-free; it can occasionally cause segmentation faults, particularly if used when finding basic solutions to underdetermined systems with many more columns than rows (ParU do not solve such systems anyway).

[*]email: aznaveh@tamu.edu. http://www.suitesparse.com.

2 Using ParU in C and C++

ParU relies on CHOLMOD for its basic sparse matrix data structure, a compressed sparse column format. CHOLMOD provides interfaces to the AMD, COLAMD, and METIS ordering methods, writing a matrix to a file, and many other functions. ParU also relies on UMFPACK Version 6.0 or higher for symbolic analysis.

2.1 Installing the C/C++ library on Linux/Unix

Before you compile the SuiteSparseQR library and demo programs, you may wish to edit the `SuiteSparse/SuiteSparse_config/SuiteSparse_config.mk` configuration file. The defaults should be fine on most Linux/Unix systems and on the Mac. It automatically detects what system you have and sets compile parameters accordingly.

Next, type `make` at the Linux/Unix command line, in either the `SuiteSparse` directory (which compiles all of SuiteSparse) or in the `SuiteSparse/SPQR` directory (which just compiles SuiteSparseQR and the libraries it requires). SuiteSparseQR will be compiled, and a set of simple demos will be run (including the one in the next section).

The configuration file defines where the LAPACK and BLAS libraries are to be found. Selecting the right BLAS is critical. There is no standard naming scheme for the name and location of these libraries. The defaults in the `SuiteSparse_config.mk` file use `-llapack` and `-lblas`; the latter may link against the standard Fortran reference BLAS, which will not provide optimal performance. For best results, you should use the OpenBLAS at openblas.net (based on the Goto BLAS) [10], or high-performance vendor-supplied BLAS such as the Intel MKL, AMD ACML, or the Sun Performance Library. Selection of LAPACK and the BLAS is done with the `LAPACK=` and `BLAS=` lines in the `SuiteSparse_config.mk` file.

Four compile-time options can be used to modify how SuiteSparseQR is compiled. Select these via the `SPQR_CONFIG=` line in the `SuiteSparse_config.mk` file.

- `-DNPARTITION`: do not compile with METIS, CAMD, or CCOLAMD. These packages are included by default.
- `-DNEXPERT`: do not compile with the “expert” routines in `SuiteSparseQR_expert.cpp`. The expert routines are included by default.
- `-DHAVE_TBB`: enable the Intel Threading Building Blocks, TBB. The use of TBB is disabled by default. It is disabled because not all installations have TBB available. The use of TBB is recommended, however, if you have a multicore computer.

To fully test 100% of the lines of SuiteSparseQR, go to the `Tcov` directory and type `make`. This will work for Linux only.

To install the shared library into /usr/local/lib and /usr/local/include, do `make install`. To uninstall, do `make uninstall`. For more options, see the `SuiteSparse/README.txt` file.

2.2 C/C++ Example

The C++ interface is written using templates for handling both real and complex matrices. The simplest function computes the MATLAB equivalent of `x=A\b` and is almost as simple:

```
#include "SuiteSparseQR.hpp"
X = SuiteSparseQR <double> (A, B, cc) ;
```

The C version of this function is almost identical:

```
#include "SuiteSparseQR_C.h"
X = SuiteSparseQR_C_backslash_default (A, B, cc) ;
```

Below is a simple C++ program that illustrates the use of SuiteSparseQR. The program reads in a least-squares problem from `stdin` in MatrixMarket format [4], solves it, and prints the norm of the residual and the estimated rank of `A`. The comments reflect the MATLAB equivalent statements. The C version of this program is identical except for the `#include` statement and call to SuiteSparseQR which are replaced with the C version of the statement above, and C-style comments.

```
#include "SuiteSparseQR.hpp"
int main (int argc, char **argv)
{
    cholmod_common Common, *cc ;
    cholmod_sparse *A ;
    cholmod_dense *X, *B, *Residual ;
    double rnorm, one [2] = {1,0}, minusone [2] = {-1,0} ;
    int mtype ;

    // start CHOLMOD
    cc = &Common ;
    cholmod_l_start (cc) ;

    // load A
    A = (cholmod_sparse *) cholmod_l_read_matrix (stdin, 1, &mtype, cc) ;

    // B = ones (size (A,1),1)
    B = cholmod_l_ones (A->nrow, 1, A->xtype, cc) ;

    // X = A\B
    X = SuiteSparseQR <double> (A, B, cc) ;

    // rnorm = norm (B-A*X)
    Residual = cholmod_l_copy_dense (B, cc) ;
    cholmod_l_sdmult (A, 0, minusone, one, X, Residual, cc) ;
    rnorm = cholmod_l_norm_dense (Residual, 2, cc) ;
    printf ("2-norm of residual: %8.1e\n", rnorm) ;
    printf ("rank %ld\n", cc->SPQR_istat [4]) ;

    // free everything and finish CHOLMOD
    cholmod_l_free_dense (&Residual, cc) ;
    cholmod_l_free_sparse (&A, cc) ;
    cholmod_l_free_dense (&X, cc) ;
    cholmod_l_free_dense (&B, cc) ;
    cholmod_l_finish (cc) ;
    return (0) ;
}
```

2.3 C++ Syntax

All features available to the MATLAB user are also available to both the C and C++ interfaces using a syntax that is not much more complicated than the MATLAB syntax. Additional features not available via the MATLAB interface include the ability to compute the symbolic and numeric factorizations separately (for multiple matrices with the same nonzero pattern but different numerical values). The following is a list of user-callable C++ functions and what they can do:

1. `SuiteSparseQR`: an overloaded function that provides functions equivalent to `spqr` and `spqr_solve` in the SuiteSparseQR MATLAB interface.

2. `SuiteSparseQR_factorize`: performs both the symbolic and numeric factorizations and returns a QR factorization object such that `A*P=Q*R`. It always exploits singletons.

3. `SuiteSparseQR_symbolic`: performs the symbolic factorization and returns a QR factorization object to be passed to `SuiteSparseQR_numeric`. It does not exploit singletons.

4. `SuiteSparseQR_numeric`: performs the numeric factorization on a QR factorization object, either one constructed by `SuiteSparseQR_symbolic`, or reusing one from a prior call to `SuiteSparseQR_numeric` for a matrix `A` with the same pattern as the first one, but with different numerical values.

5. `SuiteSparseQR_solve`: solves a linear system using the object returned by `SuiteSparseQR_factorize` or `SuiteSparseQR_numeric`, namely `x=R\b`, `x=P*R\b`, `x=R'\b`, or `x=R'\(P'*b)`.

6. `SuiteSparseQR_qmult`: provides the same function as `spqr_qmult` in the MATLAB interface, computing `Q'*x`, `Q*x`, `x*Q'`, or `x*Q`. It uses either a QR factorization in MATLAB-style sparse matrix format, or the QR factorization object returned by `SuiteSparseQR_factorize` or `SuiteSparseQR_numeric`.

7. `SuiteSparseQR_min2norm`: finds the minimum 2-norm solution to an underdetermined linear system.

8. `SuiteSparseQR_free`: frees the QR factorization object.

2.4 Details of the C/C++ Syntax

For further details of how to use the C/C++ syntax, please refer to the definitions and descriptions in the following files:

1. `SuiteSparse/SPQR/Include/SuiteSparseQR.hpp` describes each C++ function. Both `double` and `std::complex<double>` matrices are supported.

2. `SuiteSparse/SPQR/Include/SuiteSparseQR_definitions.h` describes definitions common to both C and C++ functions. For example, each of the ordering methods is given a `#define`'d name. The default is `ordering = SPQR_ORDERING_DEFAULT`, and the default tolerance is given by `tol = SPQR_DEFAULT_TOL`.

3. `SuiteSparse/SPQR/Include/SuiteSparseQR_C.h` describes the C-callable functions.

Most of the packages in SuiteSparse come in multiple versions with different sized integers. The first is the plain C/C++ `int`. The second the `SuiteSparse_long` integer, defined in the `SuiteSparse/SuiteSparse_config/SuiteSparse_config.h` file. This integer is `long` except on a Windows-64 platform for which it is the `__int64` type. The intent of `SuiteSparse_long` is that it should be 32-bits on a 32-bit platform, and 64-bits on a 64-bit platform.

By contrast, SuiteSparseQR only provides a `SuiteSparse_long` version. Most users (except Windows-64) can simply use `long` as the basic integer type passed to and returned from SuiteSparseQR.

The C/C++ options corresponding to the MATLAB `opts` parameters and the contents of the optional `info` output of `spqr_solve` are described below. Let `cc` be the CHOLMOD `Common` object, containing parameter settings and statistics. All are of type `double`, except for `SPQR_istat` which is `SuiteSparse_long`, `cc->memory_usage` which is `size_t`, and `cc->SPQR_nthreads` which is `int`. Parameters include:

`cc->SPQR_grain`	the same as `opts.grain` in the MATLAB interface
`cc->SPQR_small`	the same as `opts.small` in the MATLAB interface
`cc->SPQR_nthreads`	the same as `opts.nthreads` in the MATLAB interface

Other parameters, such as `opts.ordering` and `opts.tol`, are input parameters to the various C/C++ functions. Others such as `opts.solution='min2norm'` are separate functions in the C/C++ interface. Refer to the files listed above for details. Output statistics include:

`cc->SPQR_flopcount_bound`	an upper bound on the flop count
`cc->SPQR_tol_used`	the tolerance used (`opts.tol`)
`cc->SPQR_istat [0]`	upper bound on `nnz(R)`
`cc->SPQR_istat [1]`	upper bound on `nnz(H)`
`cc->SPQR_istat [2]`	number of frontal matrices
`cc->SPQR_istat [3]`	number of TBB tasks
`cc->SPQR_istat [4]`	estimate of the rank of `A`
`cc->SPQR_istat [5]`	number of column singletons
`cc->SPQR_istat [6]`	number of row singletons
`cc->SPQR_istat [7]`	ordering used
`cc->memory_usage`	memory used, in bytes

The upper bound on the flop count is found in the analysis phase, which ignores the numerical values of `A` (the same analysis phase operates on both real and complex matrices). Thus, if you are factorizing a complex matrix, multiply this statistic by 4.

3 GPU acceleration

As of version 2.0.0, SuiteSparseQR now includes GPU acceleration. It can exploit a single NVIDIA GPU, via CUDA. To enable GPU acceleration, you must compile SuiteSparseQR with non-default options. See the `SuiteSparse_config_GPU_gcc.mk` file in the `SuiteSparse_config` directory for details. The packages SuiteSparse_GPURuntime and GPUQREngine are also required (they should appear in the SuiteSparse directory, along with SPQR).

At run time, you must also enable the GPU by setting `Common->useGPU` to `true`. Before calling any SuiteSparseQR function, you must poll the GPU to set the available memory. Below is a sample code that initializes CHOLMOD and then polls the GPU for use in SuiteSparseQR.

```
size_t total_mem, available_mem ;
cholmod_common *cc, Common ;
cc = &Common ;
cholmod_l_start (cc) ;
cc->useGPU = true ;
cholmod_l_gpu_memorysize (&total_mem, &available_mem, cc) ;
cc->gpuMemorySize = available_mem ;
if (cc->gpuMemorySize <= 1)
{
    printf ("no GPU available\n") ;
}

// Subsequent calls to SuiteSparseQR will use the GPU, if available
```

See `Demo/qrdemo_gpu.cpp` for an extended example, which can be compiled via `make gpu` in the `Demo` directory.

GPU acceleration is not yet available via the MATLAB mexFunction interface. We expect to include this in a future release.

For a detailed technical report on the GPU-accelerated algorithm, see `qrgpu_paper.pdf` in the `Doc` directory.

4 Requirements and Availability

SuiteSparseQR requires four prior Collected Algorithms of the ACM: CHOLMOD [5, 8] (version 1.7 or later), AMD [1, 2], and COLAMD [6, 7] for its ordering/analysis phase and for its basic sparse matrix data structure, and the BLAS [9] for dense matrix computations on its frontal matrices; also required is LAPACK [3] for its Householder reflections. An efficient implementation of the BLAS is strongly recommended, either vendor-provided (such as the Intel MKL, the AMD ACML, or the Sun Performance Library) or other high-performance BLAS such as those of [10].

The use of Intel's Threading Building Blocks is optional [12], but without it, only parallelism within the BLAS can be exploited (if available). SuiteSparseQR can optionally use

METIS 4.0.1 [11] and two constrained minimum degree ordering algorithms, CCOLAMD and CAMD [5], for its fill-reducing ordering options. SuiteSparseQR can be compiled without these ordering methods and without TBB.

In addition to appearing as Collected Algorithm 8xx of the ACM, SuiteSparseQR is available at http://www.suitesparse.com and at MATLAB Central in the user-contributed File Exchange (http://www.mathworks.com/matlabcentral). See SPQR/Doc/License.txt for the license. Alternative licenses are also available; contact the author for details.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, 1996.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):381–388, 2004.

[3] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, 3rd edition, 1999.

[4] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137. Chapman & Hall, London, 1997. (`http://math.nist.gov/MatrixMarket`).

[5] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Software*, 35(3), 2009.

[6] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):377–380, 2004.

[7] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):353–376, 2004.

[8] T. A. Davis and W. W. Hager. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Trans. Math. Software*, 35(4), 2009.

[9] J. J. Dongarra, J. J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[10] K. Goto and R. van de Geijn. High performance implementation of the level-3 BLAS. *ACM Trans. Math. Software*, 35(1):4, July 2008. Article 4, 14 pages.

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

[12] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Sebastopol, CA, 2007.