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

Mohsen Aznaveh[*], Timothy A. Davis

VERSION 1.0.0, Apr XX, 2024

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 the symbolic analysis phase, after that, some symbolic analysis is done by ParU itself, and then the numeric phase starts. The numeric computation is a task parallel phase using OpenMP, and each task calls parallel BLAS; i.e. nested parallelism. The performance of BLAS has a heavy impact on the performance of ParU. Moreover, the way parallel BLAS can be called in a nested environment can also be very important for ParU's performance.

1.0.1 Instructions on using METIS

SuiteSparse uses METIS 5.1.0, distributed along with SuiteSparse itself. Its use is optional, however. ParU is using AMD as the default ordering. METIS tends to give orderings that are good for parallelism. However, the METIS itself can be slower than AMD. As a result, the symbolic analysis using METIS can be slow, but usually, the factorization is faster. Therefore, depending on your use case, either use METIS, or you can compile and run your code without using METIS. If you are using METIS on an unsymmetric case, UMFPACK has to form the Matrix A^TA. This matrix can be a dense matrix and takes a lot of resources to form it. To avoid such conditions, you can use the ordering strategy `UMFPACK_ORDERING_METIS_GUARD` that is introduced in UMFPACK version 6.0. This ordering strategy use COLAMD instead of METIS in those cases.

[*]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 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 any system

All of SuiteSparse can be built by `cmake` with a single top-level `CMakeLists.txt` file. In addition, each package (including ParU) has its own `CMakeLists.txt` file to build that package. This is the simplest method for building ParU and its dependent pacakges on all systems.

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

In Linux/MacOs, type `make` at the command line in either the `SuiteSparse` directory (which compiles all of SuiteSparse) or in the `SuiteSparse/ParU` directory (which just compiles ParU and the libraries it requires). ParU will be compiled; you can type `make demos` to run a set of simple demos.

The use of `make` is optional. The top-level `ParU/Makefile` is a simple wrapper that uses `cmake` to do the actual build.

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

To install the shared library (by default, into `/usr/local/lib` and `/usr/local/include`), do `make install`. To uninstall, do `make uninstall`. For more options, see the `ParU/README.md` file.

2.3 C/C++ Example

The C++ interface is written using only real matrices (double precision).

The simplest function computes the MATLAB equivalent of `x=A\b` and is almost as simple as MATLAB.

Below is a simple C++ program that illustrates the use of ParU. The program reads in a problem from `stdin` in MatrixMarket format [3], solves it, and prints the norm of `A` and the residual. Some error testing code is omited to simplify showing how the program works. The full program can be found in `ParU/Demo/paru_simple.cpp`

```
#include <iostream>
#include <iomanip>
#include <ios>
#include "ParU.h"

int main(int argc, char **argv)
{
```

```
    cholmod_common Common, *cc;
    cholmod_sparse *A;
    ParU_Symbolic *Sym;
    //~~~~~~~~~~Reading the input matrix and test if the format is OK~~~~~~~~~~~~
    // start CHOLMOD
    cc = &Common;
    int mtype;
    cholmod_l_start(cc);
    A = (cholmod_sparse *)cholmod_l_read_matrix(stdin, 1, &mtype, cc);
    //~~~~~~~~~~~~~~~~~~~~Starting computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    std::cout << "================= ParU, a simple demo: ==================\n";
    ParU_Control Control;
    ParU_Analyze(A, &Sym, &Control);
    std::cout << "Input matrix is " << Sym->m << "x" << Sym->n
        << " nnz = " << Sym->anz << std::endl;
    ParU_Numeric *Num;
    ParU_Factorize(A, Sym, &Num, &Control);

    //~~~~~~~~~~~~~~~~~~~~ Computing the residual, norm(b-Ax) ~~~~~~~~~~~~~~~~~~~~
    int64_t m = Sym->m;
    double *b = (double *)malloc(m * sizeof(double));
    double *xx = (double *)malloc(m * sizeof(double));
    for (int64_t i = 0; i < m; ++i) b[i] = i + 1;
    ParU_Solve(Sym, Num, b, xx, &Control);
    double resid, anorm, xnorm;
    ParU_Residual(A, xx, b, resid, anorm, xnorm, &Control);
    double rresid = (anorm == 0 || xnorm == 0 ) ? 0 : (resid/(anorm*xnorm));
    std::cout << std::scientific << std::setprecision(2)
        << "Relative residual is |" << rresid << "| anorm is " << anorm
        << ", xnorm is " << xnorm << " and rcond is " << Num->rcond << "."
        << std::endl;
    free(b);
    free(xx);

    //~~~~~~~~~~~~~~~~~~~~End computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ParU_FreeNumeric(&Num, &Control);
    ParU_FreeSymbolic(&Sym, &Control);
    cholmod_l_free_sparse(&A, cc);
    cholmod_l_finish(cc);
}
```

A simple demo for the C interface is shown next. You can see the complete demo in ParU/Demo/paru_simplec.c

```
#include "ParU.h"
int main(int argc, char **argv)
```

```
{
    cholmod_common Common, *cc;
    cholmod_sparse *A;
    ParU_C_Symbolic *Sym;
    //~~~~~~~~~Reading the input matrix and test if the format is OK~~~~~~~~~~~~
    // start CHOLMOD
    cc = &Common;
    int mtype;
    cholmod_l_start(cc);
    // A = mread (stdin) ; read in the sparse matrix A
    A = (cholmod_sparse *)cholmod_l_read_matrix(stdin, 1, &mtype, cc);
    //~~~~~~~~~~~~~~~~~~~Starting computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    printf("================= ParU, a simple demo, using C interface : ====\n");
    ParU_C_Control Control;
    ParU_C_Init_Control(&Control);
    ParU_C_Analyze(A, &Sym, &Control);
    printf("Input matrix is %" PRId64 "x%" PRId64 " nnz = %" PRId64 " \n",
        Sym->m, Sym->n, Sym->anz);
    ParU_C_Numeric *Num;
    ParU_C_Factorize(A, Sym, &Num, &Control);

    //~~~~~~~~~~~~~~~~~~~ Computing the residual, norm(b-Ax) ~~~~~~~~~~~~~~~~~~~
    int64_t m = Sym->m;
    double *b = (double *)malloc(m * sizeof(double));
    double *xx = (double *)malloc(m * sizeof(double));
    for (int64_t i = 0; i < m; ++i) b[i] = i + 1;
    ParU_C_Solve_Axb(Sym, Num, b, xx, &Control);
    double resid, anorm, xnorm;
    ParU_C_Residual_bAx(A, xx, b, &resid, &anorm, &xnorm, &Control);
    double rresid = (anorm == 0 || xnorm == 0 ) ? 0 : (resid/(anorm*xnorm));
    printf( "Relative residual is |%.2e|, anorm is %.2e, xnorm is %.2e, "
        " and rcond is %.2e.\n",
        rresid, anorm, xnorm, Num->rcond);
    free(b);
    free(xx);

    //~~~~~~~~~~~~~~~~~~End computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ParU_C_FreeNumeric(&Num, &Control);
    ParU_C_FreeSymbolic(&Sym, &Control);
    cholmod_l_free_sparse(&A, cc);
    cholmod_l_finish(cc);
}
```

2.4 C++ Syntax

2.4.1 ParU_Info: return values of each ParU method

All ParU C and C++ routines return an enum of type `ParU_Info`. The user application should check the output before continuing.

```
typedef enum ParU_Info
{
    PARU_SUCCESS = 0,           // everying is fine
    PARU_OUT_OF_MEMORY = -1,    // ParU ran out of memory
    PARU_INVALID = -2,          // inputs are invalid (NULL, for example)
    PARU_SINGULAR = -3,         // matrix is numerically singular
    PARU_TOO_LARGE = -4         // problem too large for the BLAS
} ParU_Info ;
```

2.4.2 ParU_Version: return ParU version

ParU has two mechanisms for informing the user application of its date and version: macros that are `#defined` in `ParU.h`, and a `ParU_Version` function. Both methods are provided since it's possible that the `ParU.h` header found when a user application was compiled might not match the same version found when the same user application was linked.

```
#define PARU_DATE "Apr XX, 2024"    // FIXME
#define PARU_VERSION_MAJOR  1
#define PARU_VERSION_MINOR  0
#define PARU_VERSION_UPDATE 0
ParU_Info ParU_Version (int ver [3], char date [128]) ;
```

`ParU_Version` returns the version in `ver` array (major, minor, and update, in that order), and the date in the `date` array provided by the user application.

2.4.3 ParU_Control: parameters that control ParU

The `ParU_Control` structure contains parameters that control various ParU options. When declared, the structure is initialized with default values. The user can then change the values.

ParU_Control	default value and explanation
mem_chunk	default: 2^{20}. Chunk size for parallel memset and memcpy
paru_max_threads	default: 0. Initialized with omp_max_threads
umfpack_ordering	default: UMFPACK_ORDERING_AMD. Default UMFPACK ordering
umfpack_strategy	default: UMFPACK_STRATEGY_AUTO. Default UMFPACK strategy
filter_singletons	default: 1. If nonzero, singletons are permuted to the front of the matrix before factorization. Singletons are rows or columns with a single entry (or have a single entry after other singletons are removed).
relaxed_amalgamation	default: 32. Threshold for relaxed amalgamation. When constructing its frontal matrices, ParU attempts to ensure that all frontal matrices contain at least this many pivot columns. Values less than zero are treated as 32, and values greater than 512 are treated as 512.
prescale	default: 1. 0: no scaling, 1: each row is scaled by the maximum absolute value in the row.
panel_width	default: 32. Width of panel for dense factorization
paru_strategy	default: PARU_STRATEGY_AUTO. Default strategy for ParU
piv_toler	default: 0.1. Tolerance for accepting sparse pivots
diag_toler	default: 0.001. Tolerance for accepting symmetric pivots
trivial	default: 4. Do not call BLAS for smaller dgemms
worthwhile_dgemm	default: 512. dgemms bigger than this are tasked
worthwhile_trsm	default: 4096. trsm bigger than this are tasked

The first row of the options in the table above is used in both the symbolic analysis and numerical factorization. The second row of the options is used in the symbolic analysis. The third row of control options shows those that have an impact on numerical factorization.

If set to zero, paru_max_threads is initalized by omp_max_threads.

If paru_strategy is set to PARU_STRATEGY_AUTO. ParU uses the same strategy as UMFPACK. However, the user can ask UMFPACK for an unsymmetric strategy but use a symmetric strategy for ParU. Usually, UMFPACK chooses a good ordering; however, there might be cases where users prefer unsymmetric ordering on UMFPACK but symmetric computation on ParU.

The ParU_Control structure is defined below:

```
struct ParU_Control
{
    // For all phases of ParU:
    int64_t mem_chunk = PARU_MEM_CHUNK ;  // chunk size for memset and memcpy

    // Numeric factorization parameters:
    double piv_toler = 0.1 ;    // tolerance for accepting sparse pivots
    double diag_toler = 0.001 ; // tolerance for accepting symmetric pivots
    int32_t panel_width = 32 ;  // width of panel for dense factorizaiton
    int32_t trivial = 4 ;       // dgemms smaller than this do not call BLAS
    int32_t worthwhile_dgemm = 512 ; // dgemms bigger than this are tasked
    int32_t worthwhile_trsm = 4096 ; // trsm bigger than this are tasked
    int32_t prescale = 1 ;  // 0: no scaling, 1: scale each row by the max
        // absolute value in its row.

    // Symbolic analysis parameters:
    int32_t umfpack_ordering = UMFPACK_ORDERING_METIS ;
```

```
    int32_t umfpack_strategy = UMFPACK_STRATEGY_AUTO ;
    int32_t relaxed_amalgamation = 32 ;  // symbolic analysis tries to ensure
        // that each front have more pivot columns than this threshold
    int32_t paru_strategy = PARU_STRATEGY_AUTO ;
    int32_t filter_singletons = 1 ; // filter singletons if nonzero

    // For all phases of ParU:
    int32_t paru_max_threads = 0 ;  // initialized with omp_max_threads
} ;
```

2.4.4 ParU_Analyze: symbolic analysis

```
ParU_Info ParU_Analyze
(
    // input:
    cholmod_sparse *A,  // input matrix to analyze of size n-by-n
    // output:
    ParU_Symbolic **Sym_handle,  // output, symbolic analysis
    // control:
    ParU_Control *Control
) ;
```

`ParU_Analyze` takes as input a sparse matrix in the CHOLMOD data structure, `A`. The matrix must be square and not held in the CHOLMOD symmetric storage format. Refer to the CHOLMOD documentation for details. On output, the symbolic analysis structure `Sym` is created, passed in as `&Sym`. The symbolic analysis can be used for different calls to `ParU_Factorize` for matrices that have the same sparsity pattern but different numerical values.

2.4.5 ParU_Factorize: numerical factorization

```
ParU_Info ParU_Factorize
(
    // input:
    cholmod_sparse *A,  // input matrix to factorize
    ParU_Symbolic *Sym, // symbolic analsys from ParU_Analyze
    // output:
    ParU_Numeric **Num_handle,
    // control:
    ParU_Control *Control
) ;
```

`ParU_Factorize` performs the numerical factorization of its input sparse matrix `A`. The symbolic analsys `Sym` must have been created by a prior call to `ParU_Analyze` with the same matrix `A`, or one with the same sparsity pattern as the one passed to `ParU_Factorize`. On output, the `&Num` structure is created.

2.4.6 ParU_Solve: solve a linear system, $Ax = b$

`ParU_Solve` solves a sparse linear system $Ax = b$ for a sparse matrix `A` and vectors `x` and `b`, or matrices `X` and `B`. The matrix `A` must have been factorized by `ParU_Factorize`, and the `Sym` and `Num` structures from that call must be passed to this method.

The method has four overloaded signatures, so that it can handle a single right-hand-side vector or a matrix with multiple right-hand-sides, and it provides the option of overwriting the input right-hand-side(s) with the solution(s).

```
ParU_Info ParU_Solve        // solve Ax=b, overwriting b with the solution x
(
    // input:
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    // input/output:
    double *x,              // vector of size n-by-1; right-hand on input,
                            // solution on output
    // control:
    ParU_Control *Control
) ;

ParU_Info ParU_Solve        // solve Ax=b
(
    // input:
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    double *b,              // vector of size n-by-1
    // output
    double *x,              // vector of size n-by-1
    // control:
    ParU_Control *Control
) ;

ParU_Info ParU_Solve        // solve AX=B, overwriting B with the solution X
(
    // input
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    int64_t nrhs,           // # of right-hand sides
    // input/output:
    double *X,              // X is n-by-nrhs, where A is n-by-n;
                            // holds B on input, solution X on input
    // control:
    ParU_Control *Control
) ;

ParU_Info ParU_Solve        // solve AX=B
(
    // input
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    int64_t nrhs,           // # of right-hand sides
    double *B,              // n-by-nrhs, in column-major storage
    // output:
    double *X,              // n-by-nrhs, in column-major storage
    // control:
    ParU_Control *Control
) ;
```

2.4.7 ParU_LSolve: solve $Lx = b$

`ParU_LSolve` solves a lower triangular system, $Lx = b$ with vectors x and b, or $LX = B$ with matrices X and B, using the lower triangular factor computed by `ParU_Factorize`. No scaling or permutations are used.

```
ParU_Info ParU_LSolve
(
    // input
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    // input/output:
    double *x,              // n-by-1, in column-major storage;
                            // holds b on input, solution x on input
    // control:
    ParU_Control *Control
) ;

ParU_Info ParU_LSolve
(
    // input
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    int64_t nrhs,           // # of right-hand-sides (# columns of X)
    // input/output:
    double *X,              // X is n-by-nrhs, where A is n-by-n;
                            // holds B on input, solution X on input
    // control:
    ParU_Control *Control
) ;
```

2.4.8 ParU_USolve: solve $Ux = b$

`ParU_USolve` solves a lower triangular system, $Ux = b$ with vectors x and b, or $UX = B$ with matrices X and B, using the upper triangular factor computed by `ParU_Factorize`. No scaling or permutations are used.

```
ParU_Info ParU_USolve
(
    // input
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    // input/output
    double *x,              // n-by-1, in column-major storage;
                            // holds b on input, solution x on input
    // control:
    ParU_Control *Control
) ;

ParU_Info ParU_USolve
(
    // input
    ParU_Symbolic *Sym,     // symbolic analysis from ParU_Analyze
```

```
    ParU_Numeric *Num,      // numeric factorization from ParU_Factorize
    int64_t nrhs,           // # of right-hand-sides (# columns of X)
    // input/output:
    double *X,              // X is n-by-nrhs, where A is n-by-n;
                            // holds B on input, solution X on input
    // control:
    ParU_Control *Control
) ;
```

2.4.9 ParU_Perm: permute and scale a dense vector or matrix

`ParU_Perm` permutes and optionally scales a vector b or matrix B. If the input `s` is `NULL`, no scaling is applied. The permutation vector `P` has size `n`. If the kth index in the permutation is row i, then `i = P[k]`.

For the vector case, the output is $x(k) = b(P(k))/s(P(k))$, or $x(k) = b(k)/s(k)$, or if `s` is `NULL`, for all k in the range 0 to $n-1$.

For the matrix case, the output is $X(k,j) = B(P(k),j)/s(P(k),j)$ for all rows k and all columns j of X and B. If `s` is `NULL`, then the output is $X(k,j) = B(k,j)/s(k,j)$.

```
ParU_Info ParU_Perm
(
    // inputs
    const int64_t *P,   // permutation vector of size n
    const double *s,    // vector of size n (optional)
    const double *b,    // vector of size n
    int64_t n,          // length of P, s, B, and X
    // output
    double *x,          // vector of size n
    // control:
    ParU_Control *Control
) ;

ParU_Info ParU_Perm
(
    // inputs
    const int64_t *P,   // permutation vector of size nrows
    const double *s,    // vector of size nrows (optional)
    const double *B,    // array of size nrows-by-ncols
    int64_t nrows,      // # of rows of X and B
    int64_t ncols,      // # of columns of X and B
    // output
    double *X,          // array of size nrows-by-ncols
    // control:
    ParU_Control *Control
) ;
```

2.4.10 ParU_InvPerm: permute and scale a dense vector or matrix

2.4.11 ParU_Residual: compute the residual

The `ParU_Residual` function comptes the relative residual of $Ax = b$ or $AX = B$, in the 1-norm. It also computes the 1-norm of A and the solution X or x.

```
ParU_Info ParU_Residual
(
    // inputs:
    cholmod_sparse *A,  // an n-by-n sparse matrix
    double *x,          // vector of size n
    double *b,          // vector of size n
    // output:
    double &resid,      // residual: norm1(b-A*x) / (norm1(A) * norm1 (x))
    double &anorm,      // 1-norm of A
    double &xnorm,      // 1-norm of x
    // control:
    ParU_Control *Control
) ;

ParU_Info ParU_Residual
(
    // inputs:
    cholmod_sparse *A,  // an n-by-n sparse matrix
    double *X,          // array of size n-by-nrhs
    double *B,          // array of size n-by-nrhs
    int64_t nrhs,
    // output:
    double &resid,      // residual: norm1(B-A*X) / (norm1(A) * norm1 (X))
    double &anorm,      // 1-norm of A
    double &xnorm,      // 1-norm of X
    // control:
    ParU_Control *Control
) ;
```

2.4.12 ParU_FreeNumeric: free a numeric factorization

```
ParU_Info ParU_FreeNumeric
(
    // input/output:
    ParU_Numeric **Num_handle,  // numeric object to free
    // control:
    ParU_Control *Control
) ;
```

2.4.13 ParU_FreeSymbolic: free a symbolic analysis

```
ParU_Info ParU_FreeSymbolic
(
    // input/output:
    ParU_Symbolic **Sym_handle, // symbolic object to free
    // control:
    ParU_Control *Control
) ;
```

2.5 C Syntax

The C interface is quite similar to the C++ interface, and you can see the C user callable routines in `ParU/Include/ParU.h`. The following is a list of user-callable C functions, their prototypes, and what they can do:

1. `ParU_C_Version`: return the version of the ParU package you are using.
2. `ParU_C_Init_Control`: Initialize C Control structure before using it.
3. `ParU_C_Analyze`: Symbolic analysis is done in this routine. UMFPACK is called here; after that, some more specialized symbolic computation is done for ParU. `ParU_C_Analyze` is called once and can be used for different `ParU_C_Factorize` calls for the matrices that have the same pattern.
4. `ParU_C_Factorize`: Numeric factorization is done in this routine. Scaling and making Sx (scaled and staircase structure) matrix, computing factors, and permutations are here. `ParU_C_Symbolic` structure which is computed in `ParU_C_Analyze` is an input in this routine.
5. `ParU_C_Solve_Axx`, `ParU_C_Solve_Axb`, `ParU_C_Solve_AXX` and `ParU_C_Solve_AXB`, Using symbolic analysis and factorization phase output to solve $Ax = b$. In all the solve routines Num structure must come with the same Sym struct that comes from `ParU_C_Factorize`.
6. `ParU_C_LSolve`:
7. `ParU_C_USolve`:
8. `ParU_C_Perm`:
9. `ParU_C_InvPerm`:
10. `ParU_C_Residual_bAx`: This function computes $|Ax - b|$ one norm of matrix A and one norm of x
11. `ParU_C_Residual_BAX`: This function computes $|AX - B|$ one norm of matrix A and one norme of X
12. `ParU_C_FreeNumeric`: frees the numerical part of factorization.
13. `ParU_C_FreeSymbolic`: frees the symbolic part of factorization.

2.6 Details of the C/C++ Syntax

For further details on how to use the C/C++ syntax, please refer to the definitions and descriptions in the `SuiteSparse/ParU/Include/ParU.h` file, which describes each C++ and C function. Only `double` and square matrices are supported.

FIXME: merge with subsection above on `ParU_Control`

3 Requirements and Availability

ParU requires several Collected Algorithms of the ACM: CHOLMOD [4, 7] (version 1.7 or later), AMD [1, 2], COLAMD [5, 6] and UMFPACK [8] 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. 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]. Note that while ParU uses nested parallelism heavily the right options for the BLAS library must be chosen to get a good performance.

The use of OpenMP tasking is optional, but without it, only parallelism within the BLAS can be exploited (if available). See ParU/Doc/LICENSE for the license. Alternative licenses are also available; contact the authors 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] 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`).

[4] 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.

[5] 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.

[6] 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.

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

[8] Timothy A. Davis. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, jun 2004.

[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.