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

Mohsen Aznaveh[*], Timothy A. Davis

VERSION 0.2.0, Nov 22, 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 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 is now on 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 slow. 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`

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

that is introduced in UMFPACK version 6.0. This ordering strategy use COLAMD instead of METIS in those cases.

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 does not solve such systems anyway but you might see some problems with other SuiteSparse packages.

## 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 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 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.

### 2.2 C/C++ Example

The C++ interface is written using only real matrices. The simplest function computes the MATLAB equivalent of `x=A\b` and is almost as simple: 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_demo.cpp`

```
#include "ParU.hpp"
int main(int argc, char **argv)
{
    cholmod_common Common, *cc;
    cholmod_sparse *A;
    ParU_Symbolic *Sym = NULL;

    //~~~~~~~~~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~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ParU_Control Control;
    ParU_Ret info;
    info = ParU_Analyze(A, &Sym, &Control);
    ParU_Numeric *Num;
    info = ParU_Factorize(A, Sym, &Num, &Control);
    double my_time = omp_get_wtime() - my_start_time;
    //~~~~~~~~~~~~~~~~~~~Test the results ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    int64_t m = Sym->m;
    if (info == PARU_SUCCESS)
    {
        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;
        info = ParU_Solve(Sym, Num, b, xx, &Control);
        printf("Solve time is %lf seconds.\n", my_solve_time);
        double resid, anorm, xnorm;
        info = ParU_Residual(A, xx, b, m, resid, anorm, xnorm, &Control);
        printf("Residual is |%.2e|, anorm is %.2e, xnorm is %.2e  and rcond is"
                " %.2e.\n", resid, anorm, xnorm, Num->rcond);
        free(b);
        free(xx);
    }
    //~~~~~~~~~~~~~~~~~~~End computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    int max_threads = omp_get_max_threads();
    BLAS_set_num_threads(max_threads);

    //~~~~~~~~~~~~~~~~~~~Free Everything~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ParU_Freenum(&Num, &Control);
    ParU_Freesym(&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_simple.c

```
#include "ParU_C.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_Ret info;
    info = ParU_C_Analyze(A, &Sym, &Control);
    printf("Input matrix is %ldx%ld nnz = %ld \n", Sym->m, Sym->n, Sym->anz);
    ParU_C_Numeric *Num;
    info = ParU_C_Factorize(A, Sym, &Num, &Control);

    if (info != PARU_SUCCESS)
    {
        printf("ParU: factorization was NOT successfull.");
        if (info == PARU_OUT_OF_MEMORY) printf("\nOut of memory\n");
        if (info == PARU_INVALID) printf("\nInvalid!\n");
        if (info == PARU_SINGULAR) printf("\nSingular!\n");
    }
    else
    {
        printf("ParU: factorization was successfull.\n");
    }

    //~~~~~~~~~~~~~~~~~~~ Computing Ax = b ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if (info == PARU_SUCCESS)
    {
        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;
        info = ParU_C_Solve_Axb(Sym, Num, b, xx, &Control);
        double resid, anorm, xnorm;
        info = ParU_C_Residual_bAx(A, xx, b, m, &resid, &anorm, &Control);
        printf("Residual is |%.2e|, anorm is %.2e, xnorm is %.2e  and rcond is"
               " %.2e.\n", resid, anorm, xnorm, Num->rcond);
```

```
        free(b);
        free(xx);
    }
    //~~~~~~~~~~~~~~~~~~~End computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ParU_C_Freenum(&Num, &Control);
    ParU_C_Freesym(&Sym, &Control);

    cholmod_l_free_sparse(&A, cc);
    cholmod_l_finish(cc);
}
```

## 2.3 C/C++ Syntax

`ParU_Ret` is the output structure of all ParU routines in C and C++. The user must check the output before continuing and computing further the result of prior routine. You can see the user callable routines in `ParU/Include/ParU.hpp`. The following is a list of user-callable C++ functions and what they can do:

1. `ParU_Version`: return the version of the ParU package you are using.
2. `ParU_Analyze`: Symbolic analysis is done in this routine. UMFPACK is called here and after that, some more specialized symbolic computation is done for ParU. `ParU_Analyze` is called once and can be used for different `ParU_Factorize` calls for the matrices that have the same pattern.
3. `ParU_Factorize`: Numeric factorization is done in this routine. Scaling and making $Sx$ (scaled and staircase structure) matrix, computing factors, and permutations are here. `ParU_Symbolic` structure which is computed in `ParU_Analyze` is an input in this routine.
4. `ParU_Solve`: 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_Factorize`. This routine is overloaded and can solve different systems. It has versions that keep a copy of x or overwrite it. Also, it can solve multiple right-hand side problems.
5. `ParU_Residual`: This function computes $|Ax - b|$ one norm of matrix $A$ and one norm of $x$ (or $X$ for multiple right handside).
6. `ParU_Freenum`: frees the numerical part of factorization.
7. `ParU_Freesym`: frees the symbolic part of factorization.

The C interface is quite similar to the C++ interface, and you can see the C user callable routines in `ParU/Include/ParU_C.h`. The following is a list of user-callable C functions 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 object 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_Residual_bAx`: This function computes $|Ax - b|$ one norm of matrix $A$ and one norm of $x$

7. `ParU_C_Residual_BAX`: This function computes $|AX - B|$ one norm of matrix $A$ and one norme of $X$

8. `ParU_C_Freenum`: frees the numerical part of factorization.

9. `ParU_C_Freesym`: frees the symbolic part of factorization.

## 2.4 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 following files:

1. `SuiteSparse/ParU/Include/ParU.hpp` describes each C++ function. Only `double` and square matrices are supported.

2. `SuiteSparse/ParU/Include/ParU_C.h` describes the C-callable functions.

There are C/C++ options to control ParU, which is an input argument to several routines. When you make C++ `ParU_Control` object, it is initialized with default values. The user can change the values. When using C `ParU_C_Control`, you have to fully initialize it or call `ParU_C_Init_Control` before using it.

Here is the list of control options (both in C and C++):

| ParU_Control | default value | explanation |
|---|---|---|
| mem_chunk | $1024 * 1024$ | chunk size for memset and memcpy |
| paru_max_threads | 0 | initialized with omp_max_threads |
| umfpack_ordering | UMFPACK_ORDERING_AMD | default UMFPACK ordering |
| umfpack_strategy | UMFPACK_STRATEGY_AUTO | default UMFPACK strategy |
| umfpack_default_singleton | 1 | default filter singletons if true |
| relaxed_amalgamation_threshold | 32 | threshold for relaxed amalgamation |
| scale | 1 | if 1 matrix will be scaled using max_row |
| panel_width | 32 | width of panel for dense factorizaiton |
| paru_strategy | PARU_STRATEGY_AUTO | default strategy for ParU |
| piv_toler | 0.1 | tolerance for accepting sparse pivots |
| diag_toler | 0.001 | tolerance for accepting symmetric pivots |
| trivial | 4 | Do not call BLAS for smaller dgemms |
| worthwhile_dgemm | 512 | dgemms bigger than worthwhile are tasked |
| worthwhile_trsm | 4096 | trsm bigger than worthwhile are tasked |

The first row of the options is either used in symbolic or numerical analysis. The second row of the options is used in the symbolic analysis. In the symbolic analysis phase, only the matrix pattern is probed. The third row of control options shows those that have an impact on numerical analysis.

`paru_max_threads` is initalized by `omp_max_threads` if the user do not provide a smaller number.

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.

## 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.