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.

[*]email: aznaveh@tamu.edu.

[†]email: DrTimothyAldenDavis@gmail.com, http://www.suitesparse.com.

Contents

1 Introduction

The algorithms used in ParU are discussed in a companion paper. FIXME: cite our ACM TOMS submission, and include it the Doc folder. This document gives detailed information on the installation and use of ParU. ParU is a parallel sparse direct solver that 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.

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 for its 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 individually. 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). 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 test requires Linux.

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

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 the program, but a robust user application should check the return values from ParU. The full program can be found in `ParU/Demo/paru_simple.cpp`. Note that ParU supports only real double-precision matrices.

Refer to the CHOLMOD User guide for the CHOLMOD methods used below.

```
#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 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 this return value 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 ;
```

3 C++ Syntax

3.1 ParU_Version: version of the ParU package

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 with the compiled ParU library.

```
#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.

3.2 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. Maximum number of OpenMP threads to use. If zero (the default value), this is initialized with `omp_max_threads`.
`paru_strategy`	default: `PARU_STRATEGY_AUTO`. Default ordering strategy for ParU. Options are: `PARU_STRATEGY_AUTO` (select the strategy that UMFPACK selects), `PARU_STRATEGY_UNSYMMETRIC` (order and analyze $A'A$), `PARU_STRATEGY_SYMMETRIC` (order and analyze $A + A'$).
`umfpack_strategy`	default: `UMFPACK_STRATEGY_AUTO`. Default UMFPACK strategy. Options are: `UMFPACK_STRATEGY_AUTO` (select the strategy based on how symmetric the pattern of the input matrix is), `UMFPACK_STRATEGY_UNSYMMETRIC` (order and analyze $A'A$), `UMFPACK_STRATEGY_SYMMETRIC` (order and analyze $A + A'$). Note that the `paru_strategy` and `umfpack_strategy` can differ.
`umfpack_ordering`	default: `UMFPACK_ORDERING_METIS_GUARD`. ParU relies on UMFPACK for its fill-reducing ordering; refer to the UMPFACK documentation for more details. The ordering options are defined as `UMFPACK_ORDERING_*` where `*` is one of: `METIS` (use metis on $A + A'$ or $A'A$), `METIS_GUARD` (use metis, unless $A'A$ has too many entries), `AMD` (use amd on $A + A'$ or colamd on A), `CHOLMOD` (use CHOLMOD's ordering strategy; try amd/colamd and then try METIS if the fillin from amd/colamd is high), `BEST` (use CHOLMOD to try amd/colamd, metis, and CHOLMOD's own nested dissection method, and take the best ordering found), `NONE` (no fill-reducing ordering). When using an ordering to factorize a single matrix, `AMD` is often the best since `METIS` and `METIS_GUARD` can result in low fillin but high ordering times.
`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 of each frontal matrix.
`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 section of the options in the table above is used in both the symbolic analysis and numerical factorization. The second section of the options is used in the symbolic analysis. The third section of control options shows those that have an impact on numerical factorization.

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_GUARD ;
    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
} ;
```

3.3 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. The symbolic analysis structure must be freed by `ParU_FreeSymbolic`.

3.4 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. The numeric factorization structure must be freed by ParU_FreeNumeric.

3.5 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
    ) ;
```

3.6 ParU_LSolve: solve a linear system, $Lx = b$

`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
    ) ;
```

3.7 ParU_USolve: solve a linear system, $Ux = b$

`ParU_USolve` solves an upper 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
) ;
```

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

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

```
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
) ;
```

3.9 ParU_InvPerm: permute and scale a dense vector or matrix

`ParU_InvPerm` 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`, and its inverse is implicitly used by this method. If the kth index in the permutation is row i, then `i = P[k]`.

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

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

```
ParU_Info ParU_InvPerm
(
    // 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_InvPerm
(
    // 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
) ;
```

The `ParU_LSolve`, `ParU_USolve`, `ParU_Perm`, and `ParU_InvPerm` can be used together to solve $Ax = b$ or $AX = B$. For example, if `t` is a temporary vector of size `n`, and A is an n-by-n matrix, calling `ParU_Solve` to solve $Ax = b$ is identical to the following (ignoring any tests for error conditions):

```
ParU_Perm (Num->Pfin, Num->Rs, b, n, t, Control) ;
ParU_LSolve (Sym, Num, t, Control) ;
ParU_USolve (Sym, Num, t, Control) ;
ParU_InvPerm (Sym->Qfill, NULL, t, n, x, Control) ;
```

The numeric factorization `Num` contains the row permutation vector `Num->Pfin` from partial pivoting, and the row scaling vector `Num->Rs`. The symbolic analysis structure `Sym` contains the fill-reducing column preordering, `Sym->Qfill`.

3.10 ParU_Residual: compute the residual

The `ParU_Residual` function computes 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
) ;
```

3.11 ParU_FreeNumeric: free a numeric factorization

```
ParU_Info ParU_FreeNumeric
(
    // input/output:
    ParU_Numeric *Num_handle,  // numeric object to free
```

```
        // control:
        ParU_Control *Control
    ) ;
```

3.12 ParU_FreeSymbolic: free a symbolic analysis

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

4 C Syntax

The C interface is quite similar to the C++ interface. The next sections describe the user-callable C functions, their prototypes, and what they can do.

4.1 ParU_C_Version: version of the ParU package

```
    ParU_Info ParU_C_Version (int ver [3], char date [128]) ;
```

4.2 ParU_C_Init_Control: sets control parameters to defaults

```
    ParU_Info ParU_C_Init_Control (ParU_C_Control *Control_C) ;
```

4.3 ParU_C_Analyze: symbolic analysis

`ParU_C_Analyze` performs the symbolic analysis of a sparse matrix, based solely on its nonzero pattern. `ParU_C_Analyze` is called once and can be used for different `ParU_C_Factorize` calls for the matrices that have the same pattern but different numerical values. The symbolic analysis structure must be freed by `ParU_C_FreeSymbolic`.

```
    ParU_Info ParU_C_Analyze
    (
        // input:
        cholmod_sparse *A,  // input matrix to analyze of size n-by-n
        // output:
        ParU_C_Symbolic *Sym_handle_C,  // output, symbolic analysis
        // control:
        ParU_C_Control *Control_C
    ) ;
```

4.4 ParU_C_Factorize: numeric factorization

`ParU_C_Factorize` computes the numeric factorization. The `ParU_C_Symbolic` structure computed in `ParU_C_Analyze` is an input to this routine. The numeric factorization structure must be freed by `ParU_C_FreeNumeric`.

```
ParU_Info ParU_C_Factorize
(
    // input:
    cholmod_sparse *A,          // input matrix to factorize of size n-by-n
    ParU_C_Symbolic Sym_C,      // symbolic analysis from ParU_Analyze
    // output:
    ParU_C_Numeric *Num_handle_C,    // output numerical factorization
    // control:
    ParU_C_Control *Control_C
) ;
```

4.5 ParU_C_Solve_A*: solve a linear system, $Ax = b$

The `ParU_C_Solve_Axx`, `ParU_C_Solve_Axb`, `ParU_C_Solve_AXX` and `ParU_C_Solve_AXB` methods solve 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.

```
ParU_Info ParU_C_Solve_Axx
(
    // input:
    ParU_C_Symbolic Sym_C,  // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    // input/output:
    double *x,              // vector of size n-by-1; right-hand on input,
                            // solution on output
    // control:
    ParU_C_Control *Control_C
) ;

ParU_Info ParU_C_Solve_Axb
(
    // input:
    ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    double *b,              // vector of size n-by-1
    // output
    double *x,              // vector of size n-by-1
    // control:
    ParU_C_Control *Control_C
) ;

ParU_Info ParU_C_Solve_AXX
(
    // input
    ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    int64_t nrhs,
    // input/output:
    double *X,              // array of size n-by-nrhs in column-major storage,
                            // right-hand-side on input, solution on output.
    // control:
    ParU_C_Control *Control_C
) ;
```

```
ParU_Info ParU_C_Solve_AXB
(
    // input
    ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    int64_t nrhs,
    double *B,              // array of size n-by-nrhs in column-major storage
    // output:
    double *X,              // array of size n-by-nrhs in column-major storage
    // control:
    ParU_C_Control *Control_C
) ;
```

4.6 ParU_C_Solve_L*: solve a linear system, $Lx = b$

The `ParU_C_Solve_Lxx` and `ParU_C_Solve_LXX` methods solve lower triangular systems, $Lx = b$ 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_C_Solve_Lxx
(
    // input:
    ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    // input/output:
    double *x,              // vector of size n-by-1; right-hand on input,
                            // solution on output
    // control:
    ParU_C_Control *Control_C
) ;

ParU_Info ParU_C_Solve_LXX
(
    // input
    ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    int64_t nrhs,
    // input/output:
    double *X,              // array of size n-by-nrhs in column-major storage,
                            // right-hand-side on input, solution on output.
    // control:
    ParU_C_Control *Control_C
) ;
```

4.7 ParU_C_Solve_U*: solve a linear system, $Ux = b$

The `ParU_C_Solve_Uxx` and `ParU_C_Solve_UXX` methods solve an upper triangular system, $Ux = b$ or $UX = B$. No scaling or permutation is performed.

```
ParU_Info ParU_C_Solve_Uxx
(
    // input:
```

```
    ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    // input/output:
    double *x,              // vector of size n-by-1; right-hand on input,
                            // solution on output
    // control:
    ParU_C_Control *Control_C
) ;

ParU_Info ParU_C_Solve_UXX
(
    // input
    ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    int64_t nrhs,
    // input/output:
    double *X,              // array of size n-by-nrhs in column-major storage,
                            // right-hand-side on input, solution on output.
    // control:
    ParU_C_Control *Control_C
) ;
```

4.8 ParU_C_Perm: permute and scale a dense vector or matrix

ParU_C_Perm and ParU_C_Perm_X permutes and optionally scale a dense vector or matrix. Refer to Section 3.8 for details.

```
ParU_Info ParU_C_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_C_Control *Control_C
) ;

ParU_Info ParU_C_Perm_X
(
    // 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_C_Control *Control_C
) ;
```

4.9 ParU_C_InvPerm: permute and scale a dense vector or matrix

`ParU_C_InvPerm` and `ParU_C_InvPerm_X` and permutes and optionally scale a dense vector or matrix. Refer to Section 3.9 for details.

```
ParU_Info ParU_C_InvPerm
(
    // 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_C_Control *Control_C
) ;

ParU_Info ParU_C_InvPerm_X
(
    // 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_C_Control *Control_C
) ;
```

4.10 ParU_C_Residual_*: compute the residual

`ParU_C_Residual_bAx` and `ParU_C_Residual_BAX` compute the relative residual of $Ax = b$ or $AX = B$, in the 1-norm, and the 1-norm of A and the solution X or x.

```
ParU_Info ParU_C_Residual_bAx
(
    // inputs:
    cholmod_sparse *A,  // an n-by-n sparse matrix
    double *x,          // vector of size n
    double *b,          // vector of size n
    // output:
    double *residc,     // residual: norm1(b-A*x) / (norm1(A) * norm1 (x))
    double *anormc,     // 1-norm of A
    double *xnormc,     // 1-norm of x
    // control:
    ParU_C_Control *Control_C
) ;

ParU_Info ParU_C_Residual_BAX
(
    // 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 *residc,     // residual: norm1(B-A*X) / (norm1(A) * norm1 (X))
    double *anormc,     // 1-norm of A
    double *xnormc,     // 1-norm of X
    // control:
    ParU_C_Control *Control_C
) ;
```

4.11 ParU_C_FreeNumeric: free the numeric factorization

```
    ParU_Info ParU_C_FreeNumeric
    (
        ParU_C_Numeric *Num_handle_C,    // numeric object to free
        // control:
        ParU_C_Control *Control_C
    ) ;
```

4.12 ParU_C_FreeSymbolic: free the symbolic analysis structure

```
    ParU_Info ParU_C_FreeSymbolic
    (
        ParU_C_Symbolic *Sym_handle_C,   // symbolic object to free
        // control:
        ParU_C_Control *Control_C
    ) ;
```

4.13 ParU_Get: get statistics and LU factorization properties

FIXME add `Paru_Get` to the user guide

5 Thread safety of malloc, calloc, realloc, and free

ParU is a C++ library but uses the C memory manager for all of its memory allocations, for compatibility with the other packages in SuiteSparse. It makes limited use of the C++ `new` and `delete`, but overrides those functions to use `SuiteSparse_malloc` and `SuiteSparse_free`. ParU relies on the memory manager routines defined by the `SuiteSparse_config` library (`SuiteSparse_malloc`, `SuiteSparse_calloc`, `SuiteSparse_realloc`, and `SuiteSparse_free`). By default, those routines relies on the C `malloc`, `calloc`, `realloc`, and `free` methods, respectively. They can be redefined; refer to the documentation of `SuiteSparse_config` on how to do this.

The `malloc`, `calloc`, `realloc`, and `free` methods must be thread-safe, since ParU calls those methods from within its parallel tasks. All of their implementations in the standard

C libraries that we are aware of are thread-safe. However, if your memory manager routines are not thread-safe, ParU will fail catastrophically.

6 Using ParU in MATLAB

FIXME: describe paru mexFunction: how to compile and use it. Discuss the impact of the Intel MKL BLAS, and parallel tasking, on its performance. State that it uses malloc, not mxMalloc, except for results it returns to MATLAB.

7 Requirements and Availability

ParU requires several Collected Algorithms of the ACM: CHOLMOD [4, 7], 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, such as the Intel MKL, the AMD ACML, OpenBLAS, FLAME [10]. or vendor-provide BLAS. Note that while ParU uses nested parallelism heavily the right options for the BLAS library must be chosen to get a good performance.

FIXME: describe the impact of MKL BLAS on ParU's performance, and parallel tasking.

SuiteSparse uses a slightly modified version of METIS 5.1.0, distributed along with SuiteSparse itself. Its use is optional, however. ParU uses AMD as its default ordering. METIS tends to give orderings that are good for parallelism. However, 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 must form the Matrix $A^T A$. This matrix can have many entries it takes a lot of resources to form it. To avoid such conditions, ParU uses the ordering strategy `UMFPACK_ORDERING_METIS_GUARD` by default. This ordering strategy uses COLAMD instead of METIS in when $A^T A$ is too costly to construct.

This modified version of METIS is built into CHOLMOD itself, with all functions renamed, so it does not conflict with a standard METIS library. The unmodified METIS library can be safely linked with an application that uses the modified METIS inside CHOLMOD, without any linking conflicts.

The use of OpenMP tasking is optional, but without it, only parallelism within the BLAS can be exploited (if available). ParU depends on parallel tasking to factorize multiple fronts at the same time, and performance will suffer if the compiler and BLAS library are not suitable for this method.

See `ParU/LICENSE.txt` 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.