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

Mohsen Aznaveh[*], Timothy A. Davis[†]

VERSION 1.0.0, Sept 5, 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 [3], a copy of which is in the `ParU/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 [4], 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 <cmath>
#include "ParU.h"

int main(int argc, char **argv)
{
    cholmod_common Common, *cc;
    cholmod_sparse *A = NULL;
    ParU_Symbolic Sym = NULL;
    ParU_Numeric Num = NULL;
    ParU_Control Control = NULL;
    double *b = NULL, *x = NULL;

    //~~~~~~~~~~Reading the input matrix ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // 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_Info info;
    ParU_Analyze(A, &Sym, Control);
    int64_t n, anz;
    ParU_Get (Sym, Num, PARU_GET_N, &n, Control));
    ParU_Get (Sym, Num, PARU_GET_ANZ, &anz, Control));
    std::cout << "Input matrix is " << n << "x" << n <<
        " nnz = " << anz << std::endl;
    ParU_Factorize(A, Sym, &Num, Control);
    std::cout << "ParU: factorization was successful." << std::endl;

    //~~~~~~~~~~~~~~~~~~~~ Computing the residual, norm(b-Ax) ~~~~~~~~~~~~~~~~~~~
    b = (double *)malloc(n * sizeof(double));
    x = (double *)malloc(n * sizeof(double));
    for (int64_t i = 0; i < n; ++i) b[i] = i + 1;
    ParU_Solve(Sym, Num, b, x, Control));
    double resid, anorm, xnorm, rcond;
```

```
    ParU_Residual(A, x, b, resid, anorm, xnorm, Control));
    ParU_Get (Sym, Num, PARU_GET_RCOND_ESTIMATE, &rcond, 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 " << rcond << "."
        << std::endl;

    //~~~~~~~~~~~~~~~~~~~End computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    free(b);
    free(x);
    ParU_FreeNumeric(&Num, Control);
    ParU_FreeSymbolic(&Sym, Control);
    ParU_FreeControl(&Control);
    cholmod_l_free_sparse(&A, cc);
    cholmod_l_finish(cc);
    return (info);
}
```

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 = NULL;
    cholmod_sparse *A = NULL;
    ParU_C_Symbolic Sym = NULL;
    ParU_C_Numeric Num = NULL;
    ParU_C_Control Control = NULL;
    double *b = NULL, *x = NULL;

    //~~~~~~~~~Reading the input matrix ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // start CHOLMOD
    cc = &Common;
    int mtype;
    cholmod_l_start(cc);
    // read in the sparse matrix A from stdin
    A = (cholmod_sparse *)cholmod_l_read_matrix(stdin, 1, &mtype, cc);

    //~~~~~~~~~~~~~~~~~~~Starting computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ParU_C_Analyze(A, &Sym, Control);
    int64_t n, anz;
    ParU_C_Get_INT64 (Sym, Num, PARU_GET_N, &n, Control);
    ParU_C_Get_INT64 (Sym, Num, PARU_GET_ANZ, &anz, Control);
```

```
    printf("Input matrix is %" PRId64 "x%" PRId64 " nnz = %" PRId64 " \n",
        n, n, anz);
    ParU_C_Factorize(A, Sym, &Num, Control);

    //~~~~~~~~~~~~~~~~~~~ Computing the residual, norm(b-Ax) ~~~~~~~~~~~~~~~~~~~
    b = (double *)malloc(n * sizeof(double));
    x = (double *)malloc(n * sizeof(double));
    for (int64_t i = 0; i < n; ++i) b[i] = i + 1;
    ParU_C_Solve_Axb(Sym, Num, b, x, Control);
    double resid, anorm, xnorm;
    ParU_C_Residual_bAx(A, x, b, &resid, &anorm, &xnorm, Control);
    double rresid = (anorm == 0 || xnorm == 0 ) ? 0 : (resid/(anorm*xnorm));
    double rcond;
    ParU_C_Get_FP64(Sym, Num, PARU_GET_RCOND_ESTIMATE, &rcond, Control);
    printf("Relative residual is |%.2e|, anorm is %.2e, xnorm is %.2e, "
        " and rcond is %.2e.\n",
        rresid, anorm, xnorm, rcond);

    //~~~~~~~~~~~~~~~~~~~End computation~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if (b != NULL) free(b);
    if (x != NULL) free(x);
    ParU_C_FreeNumeric(&Num, Control);
    ParU_C_FreeSymbolic(&Sym, Control);
    ParU_C_FreeControl(&Control);
    cholmod_l_free_sparse(&A, cc);
    cholmod_l_finish(cc);
    return (info);
}
```

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 "Sept 5, 2024"
#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. The object is created by `ParU_InitControl`, modified by `ParU_Set`, and deleted by `ParU_FreeControl`. Its contents can be queried with `ParU_Get`.

Any ParU function can be passed a NULL pointer for its `Control` parameter. In that case, defaults are used. To use non-default parameters, create a `Control` object and then set its parameters.

```
ParU_Info ParU_InitControl
(
    // output:
    ParU_Control *Control_handle    // Control object to create
) ;

ParU_Info ParU_FreeControl
(
    // input/output:
    ParU_Control *Control_handle    // Control object to free
) ;
```

3.3 ParU_Set: set parameters in the Control object

There are four variants of `ParU_Set` for setting control parameters, two for integers (`int64_t` and `int32_t`) and two for floating-point (`double` and `float`) which are the valid options for `type` in the signature below.

```
ParU_Info ParU_Set
(
    // input
```

```
    ParU_Control_enum parameter,    // parameter to set
    type c,                         // value to set it to
    // control:
    ParU_Control Control
) ;
```

The `ParU_Control_enum parameter` defines which parameter to set, described below. These are also used for `ParU_Get`, to read back these parameters from the `Control` object.

```
// enum for ParU_Set/ParU_Get for Control object
typedef enum
{

    // int64_t parameters for ParU_Set and ParU_Get:
    PARU_CONTROL_MAX_THREADS = 1001,          // max number of threads
    PARU_CONTROL_STRATEGY = 1002,             // ParU strategy
    PARU_CONTROL_UMFPACK_STRATEGY = 1003,     // UMFPACK strategy
    PARU_CONTROL_ORDERING = 1004,             // UMFPACK ordering
    PARU_CONTROL_RELAXED_AMALGAMATION = 1005, // goal for # pivots in fronts
    PARU_CONTROL_PANEL_WIDTH = 1006,          // # of pivots in a panel
    PARU_CONTROL_DGEMM_TINY = 1007,           // dimension of tiny dgemm's
    PARU_CONTROL_DGEMM_TASKED = 1008,         // dimension of tasked dgemm's
    PARU_CONTROL_DTRSM_TASKED = 1009,         // dimension of tasked dtrsm's
    PARU_CONTROL_PRESCALE = 1010,             // prescale input matrix
    PARU_CONTROL_SINGLETONS = 1011,           // filter singletons, or not
    PARU_CONTROL_MEM_CHUNK = 1012,            // chunk size of memset and memcpy

    // int64_t parameter, for ParU_Get only:
    PARU_CONTROL_OPENMP = 1013,               // if ParU compiled with OpenMP;
                                              // (for ParU_Get only, not set)
    PARU_CONTROL_NUM_THREADS = 1014,          // actual number of threads used

    // double parameters for ParU_Set and ParU_Get:
    PARU_CONTROL_PIVOT_TOLERANCE = 2001,      // pivot tolerance
    PARU_CONTROL_DIAG_PIVOT_TOLERANCE = 2002, // diagonal pivot tolerance

    // pointer to const string (const char **), for ParU_Get only:
    PARU_CONTROL_BLAS_LIBRARY_NAME = 3001,    // BLAS library used
    PARU_CONTROL_FRONT_TREE_TASKING = 3002,   // parallel or sequential

}
ParU_Control_enum ;
```

For integer parameters:

- `PARU_CONTROL_MAX_THREADS`: number of OpenMP threads to use. If zero or negative, the value is obtained from `omp_get_max_threads`.
- `PARU_CONTROL_STRATEGY`: Ordering and factorization strategy to use.
  - `PARU_STRATEGY_AUTO`: ParU selects its strategy automatically, based on the symbolic analysis of the input matrix, by selecting whichever strategy that UMFPACK selects.

  - `PARU_STRATEGY_UNSYMMETRIC`: During numerical factorization, no preference is given for diagonal entries when looking for pivots.
  - `PARU_STRATEGY_SYMMETRIC`: During numerical factorization, diagonal entries are given preference when looking for pivots. This strategy works well when the nonzero pattern of the matrix is mostly symmetric, and when the diagonal of the matrix is mostly zero-free.

- `PARU_CONTROL_UMFPACK_STRATEGY`: The ordering strategy used by UMFPACK. ParU uses UMFPACK for its ordering and symbolic analysis phases. The ParU and UMFPACK strategies are normally the same, but there are cases where best performance is obtained with different strategies.

  - `UMFPACK_STRATEGY_AUTO`: UMFPACK selects its strategy automatically, based on the symbolic analysis of the input matrix. Let S be the matrix found by UMFPACK after it removes the row and column *singletons* (defined below). If the singleton removal preserves the diagonal of A, the nonzero pattern of S has a *symmetry* $\sigma \geq 0.3$, and the diagonal of S is at least 90% nonzero, then the symmetric strategy is chosen. Otherwise, the unsymmetric strategy is chosen.

    The *symmetry* σ of S is defined as the number of *matched* off-diagonal entries, divided by the total number of off-diagonal entries. An entry s_{ij} is matched if s_{ji} is also an entry. They need not be numerically equal. An *entry* is a value in A which is present in the input data structure. All nonzeros are entries, but some entries may be numerically zero.

    A *row singleton* is an entry a_{ij} with a single entry in the ith row of the matrix A. A *column singleton* is an entry a_{ij} with a single entry in the jth column of the matrix A. When a singleton a_{ij} is found, row i and column j are removed and the process repeats. In the final pruned matrix, all rows and columns have at least two entries.
  - `UMFPACK_STRATEGY_UNSYMMETRIC`: UMFPACK will order columns of the matrix $A'A$ via COLAMD or METIS.
  - `UMFPACK_STRATEGY_SYMMETRIC`: UMFPACK will order the columns of the matrix $A + A'$ via AMD or METIS.

- `PARU_CONTROL_ORDERING`:

  The default ordering is `PARU_ORDERING_METIS_GUARD`, which provides low fill-in. However, this ordering can be costly to compute. It is best suited to the case when multiple matrices with the same nonzero pattern are being factorized, where the symbolic analysis is just performed once, and reused for each of the subsequent numerical factorizations.

  For a one-off factorization of a single matrix, `PARU_ORDERING_AMD` can be faster; the ordering is much faster to compute than METIS, and the quality of the ordering (which determines the fill-in and flop count in the numerical factorization) can often be acceptable. This ordering option uses AMD when using the symmetric strategy, or COLAMD when using the unsymmetric strateg.

  - `PARU_ORDERING_AMD`: use AMD on $A + A'$ (symmetric strategy) or COLAMD (unsymmetric strategy), which orders $A'A$ without forming it explicitly.
  - `PARU_ORDERING_METIS`: use METIS on $A + A'$ (symmetric strategy) or $A'A$ (unsymmetric strategy), where $A'A$ is explicitly formed.
  - `PARU_ORDERING_METIS_GUARD`: use METIS, AMD, or COLAMD. This is the default. Symmetric strategy: always use METIS on $A + A'$. Unsymmetric strategy: use METIS on $A'A$, unless A has one or more rows with $3.2\sqrt{n}$ or more entries. In that case, $A'A$ is very costly to form, and COLAMD is used instead of METIS.
  - `PARU_ORDERING_CHOLMOD`: use CHOLMOD (AMD/COLAMD then METIS, see above).
  - `PARU_ORDERING_BEST`: try many orderings and pick the best one found.
  - `PARU_ORDERING_NONE`: natural ordering. The permutations P and Q are identity, unless singletons are removed prior to factorization.

- `PARU_CONTROL_RELAXED_AMALGAMATION`: 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 the default (32), and values greater than 512 are treated as 512.

- `PARU_CONTROL_PANEL_WIDTH`: Width of panel for dense factorization of each frontal matrix.

- `PARU_CONTROL_DGEMM_TINY`: Do not call the BLAS `dgemm` routine if all dimenions of its dense matrices are small than this threshold.

- `PARU_CONTROL_DGEMM_TASKED`: When calling `dgemm`, if any dimension of its matriices are at or above this threshold, then a tasked variant of `dgemm` is used. For the Intel MKL BLAS library, this is a standard call to `dgemm`, controlled by `mkl_set_num_threads_local`. For other BLAS library, ParU makes multiple calls to `dgemm` using a single thread each.

- `PARU_CONTROL_DTRSM_TASKED`: When calling `dtrsm`, if any dimension of its matriices are at or above this threshold, then a tasked variant of `dtrsm` is used. For the Intel MKL BLAS library, this is a standard call to `dtrsm`, controlled by `mkl_set_num_threads_local`. For other BLAS library, ParU makes multiple calls to `dtrsm` using a single thread each.

- `PARU_CONTROL_PRESCALE`:

  - `PARU_PRESCALE_MAX`: each row is scaled by the maximum absolute value in the row.
  - `PARU_PRESCALE_SUM`: each row is scaled by the sum of absolute values in the row.
  - `PARU_PRESCALE_NONE`: no scaling is performed.

- `PARU_CONTROL_SINGLETONS`: If nonzero, singletons are permuted to the front of the matrix before factorization. If zero, singletons are left as-is and not treated specially.

- PARU_CONTROL_MEM_CHUNK: chunk size for parallel memset and memcpy.

For double parameters:

- PARU_CONTROL_PIVOT_TOLERANCE: Pivot tolerance for off-diagonal pivots, or for all pivots when using the unsymmetric strategy. A pivot is chosen if it is at least as large as 0.1 (default) times the maximum absolute value in its column. This threshold allows for the selection of sparse pivot rows. Standard partial pivoting is used with the tolerance is 1.0.

- PARU_CONTROL_DIAG_PIVOT_TOLERANCE: Pivot tolerance for diagonal pivots when using the symmetric strategy.

Default values of Control parameters are defined below:

```
#define PARU_DEFAULT_MAX_THREADS            (0)
#define PARU_DEFAULT_STRATEGY               PARU_STRATEGY_AUTO
#define PARU_DEFAULT_UMFPACK_STRATEGY       UMFPACK_STRATEGY_AUTO
#define PARU_DEFAULT_ORDERING               PARU_ORDERING_METIS_GUARD
#define PARU_DEFAULT_RELAXED_AMALGAMATION   (32)
#define PARU_DEFAULT_PANEL_WIDTH            (32)
#define PARU_DEFAULT_DGEMM_TINY             (4)
#define PARU_DEFAULT_DGEMM_TASKED           (512)
#define PARU_DEFAULT_DTRSM_TASKED           (4096)
#define PARU_DEFAULT_PRESCALE               PARU_PRESCALE_MAX
#define PARU_DEFAULT_SINGLETONS             (1)
#define PARU_DEFAULT_MEM_CHUNK              (1024*1024)
#define PARU_DEFAULT_PIVOT_TOLERANCE        (0.1)
#define PARU_DEFAULT_DIAG_PIVOT_TOLERANCE   (0.001)
```

Refer to the next section for how to use ParU_Get to query the current settings of parameters in the Control object.

3.4 ParU_Get: get information from a ParU opaque object

The ParU_Get method returns properties fron any of the three opaque ParU data structures: the ParU_Control object, the ParU_Symbolic object containing a symbolic analysis, and the ParU_Numeric object containing a numeric factorization.

There are several signatures for ParU_Get depending on which object is being queried. To query the ParU_Control object, the ParU_Control_enum is used (see Section 3.3). To query the other two objects (ParU_Symbolic and ParU_Numeric), the ParU_Get_enum is used, described below.

```
// enum for ParU_Get for Symbolic/Numeric objects
typedef enum
{
    // int64_t scalars:
    PARU_GET_N = 0,                 // # of rows/columns of A and its factors
    PARU_GET_ANZ = 1,               // # of entries in input matrix
    PARU_GET_LNZ_BOUND = 2,         // # of entries held in L
```

```
    PARU_GET_UNZ_BOUND = 3,         // # of entries held in U
    PARU_GET_NROW_SINGLETONS = 4,   // # of row singletons
    PARU_GET_NCOL_SINGLETONS = 5,   // # of column singletons
    PARU_GET_STRATEGY = 6,          // strategy used by ParU
    PARU_GET_UMFPACK_STRATEGY = 7,  // strategy used by UMFPACK
    PARU_GET_ORDERING = 8,          // ordering used by UMFPACK

    // int64_t arrays of size n:
    PARU_GET_P = 101,               // partial pivoting row ordering
    PARU_GET_Q = 102,               // fill-reducing column ordering

    // double scalars:
    PARU_GET_FLOPS_BOUND = 201,     // flop count for factorization (bound)
    PARU_GET_RCOND_ESTIMATE = 202,  // rcond estimate
    PARU_GET_MIN_UDIAG = 203,       // min (abs (diag (U)))
    PARU_GET_MAX_UDIAG = 204,       // max (abs (diag (U)))

    // double array of size n:
    PARU_GET_ROW_SCALE_FACTORS = 301,   // row scaling factors
}
ParU_Get_enum ;
```

Use the following signatures to query the symbolic or numeric factorization:

```
ParU_Info ParU_Get              // get int64_t from the symbolic/numeric objects
(
    // input:
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const ParU_Numeric Num,     // numeric factorization from ParU_Factorize
    ParU_Get_enum field,        // field to get
    // output:
    int64_t *result,            // int64_t result: a scalar or an array
    // control:
    ParU_Control Control
) ;

ParU_Info ParU_Get              // get double from the symbolic/numeric objects
(
    // input:
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const ParU_Numeric Num,     // numeric factorization from ParU_Factorize
    ParU_Get_enum field,        // field to get
    // output:
    double *result,             // double result: a scalar or an array
    // control:
    ParU_Control Control
) ;
```

These fields are described below.

- `PARU_GET_N`: the number of rows and columns of the matrices A, L, and U.
- `PARU_GET_ANZ`: the number of entries in the input matrix A.

- PARU_GET_LNZ_BOUND: the number of entries held in the data structure for L. This is an upper bound on the number of nonzeros in L, because it includes extra zeros due to amalgamation.
- PARU_GET_UNZ_BOUND: the number of entries held in the data structure for U. This is an upper bound on the number of nonzeros in U, because it includes extra zeros due to amalgamation.
- PARU_GET_NROW_SINGLETONS: number of row singletons, which are rows of A with just a single entry (or become so when other singletons are removed). These become diagonal pivot entries, where the corresponding row of U has just a single entry.
- PARU_GET_NCOL_SINGLETONS: number of column singletons, which are columns of A with just a single entry (or become so when other singletons are removed). These become diagonal pivot entries, where the corresponding column of L has just a single entry.
- PARU_GET_STRATEGY: the strategy selected by ParU, either PARU_STRATEGY_UNSYMMETRIC or PARU_STRATEGY_SYMMETRIC.
- PARU_GET_UMFPACK_STRATEGY: the strategy selected by UMFPACK for the symbolic analysis phase of ParU, either UMFPACK_STRATEGY_UNSYMMETRIC or UMFPACK_STRATEGY_SYMMETRIC.
- PARU_GET_ORDERING: The ordering used during the symbolic analysis phase of ParU. See the list in Section 3.3 under the description of PARU_CONTROL_ORDERING.
- PARU_GET_P: partial pivoting row ordering, an int64_t array of size n where A is n-by-n.
- PARU_GET_Q: fill-reducing column ordering, an int64_t array of size n where A is n-by-n.
- PARU_GET_FLOPS_BOUND: an upper bound on the number of floating-operations performed to compute the LU factorization. This includes extra flops due to amalgamation of frontal matrices. It does not include the prescaling of A, which takes an additional anz flops (see PARU_GET_ANZ).
- PARU_GET_RCOND_ESTIMATE: a rough estimate of the reciprical of the condition number of A, equal to the minimum absolute value on the diagonal of U, divided by the maximum absolute value on the diagonal of U.
- PARU_GET_MIN_UDIAG: the minimum absolute value on the diagonal of U.
- PARU_GET_MAX_UDIAG: the maximum absolute value on the diagonal of U.
- PARU_GET_ROW_SCALE_FACTORS: The row scaling factors, a double array of size n.

For example, to get a copy of the size-n column permutation vector from the Symbolic object:

```
int64_t Q [n] ;
ParU_Get (Sym, Num, PARU_GET_Q, Q, Control) ;
```

Most of the `int64_t` results can be obtained with a NULL numeric object, with the exception of the row permutation `P`, and the count of the number of entries in the `L` and `U` factors. All of the `double` results require both the `Sym` and `Num` objects to be valid.

The following three signatures are available for querying contents of the Control object:

```
ParU_Info ParU_Get              // get int64_t parameter from Control
(
    // input
    ParU_Control_enum field,    // field to get
    // output:
    int64_t *c,                 // value of field
    // control:
    ParU_Control Control
) ;

ParU_Info ParU_Get              // get double parameter from Control
(
    // input
    ParU_Control_enum field,    // field to get
    // output:
    double *c,                  // value of field
    // control:
    ParU_Control Control
) ;

ParU_Info ParU_Get              // get string from Control
(
    // input:
    ParU_Control_enum field,    // field to get
    // output:
    const char **result,        // string result
    // control:
    ParU_Control Control
) ;
```

The parameters that can be returned are the same as those described in Section 3.3, with four additional parameters that can be queried by `ParU_Get` but not set with `ParU_Set`. Two cases return string that is owned by the library and must not be modified:

```
char *blas_libary_name, *front_tasking ;
ParU_Get (PARU_CONTROL_BLAS_LIBRARY_NAME, &blas_name, Control)) ;
ParU_Get (PARU_CONTROL_FRONT_TREE_TASKING, &front_tasking, Control)) ;
```

This case returns true if ParU was compiled with OpenMP, or false otherwise:

```
int64_t openmp_used ;
ParU_Get (PARU_CONTROL_OPENMP, &openmp_used, Control)) ;
```

Finally, the number of threads actually to be used by ParU can be queried as follows. If the `Control` is set to its default, this will be the value from `omp_get_max_threads`. Otherwise, the value is smaller of `omp_get_max_threads` and `PARU_CONTROL_MAX_THREADS`.

```
int64_t nthreads_used ;
ParU_Get (PARU_CONTROL_NUM_THREADS, &nthreads_used, Control)) ;
```

3.5 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, double precision, not complex, 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`.

3.6 ParU_Factorize: numerical factorization

```
ParU_Info ParU_Factorize
(
    // input:
    cholmod_sparse *A,          // input matrix to factorize
    const ParU_Symbolic Sym,    // symbolic analysis 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`.

3.7 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 solution x
(
    // input:
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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:
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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 solution X
(
    // input
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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
) ;
```

3.8 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           // solve Lx=b
(
    // input
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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           // solve LX=B
(
    // input
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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
) ;
```

3.9 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           // solve Ux=b
(
    // input
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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           // solve UX=B
(
    // input
    const ParU_Symbolic Sym,    // symbolic analysis from ParU_Analyze
    const 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
) ;
```

3.10 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
) ;
```

3.11 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):

```
int64_t P [n], Q [n] ;
double t [n], R [n] ;
ParU_Get (Sym, Num, PARU_GET_P, P, Control) ;
ParU_Get (Sym, Num, PARU_GET_Q, Q, Control) ;
ParU_Get (Sym, Num, PARU_GET_ROW_SCALE_FACTORS, R, Control) ;
ParU_Perm (P, R, b, n, t, Control) ;
ParU_LSolve (Sym, Num, t, Control) ;
ParU_USolve (Sym, Num, t, Control) ;
ParU_InvPerm (Q, NULL, t, n, x, Control) ;
```

The numeric factorization `Num` contains the row permutation vector `P` from partial pivoting, and the row scaling vector `R`. The symbolic analysis structure `Sym` contains the fill-reducing column preordering, `Q`.

3.12 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, solution to Ax=b
    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, solution to AX=B
    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
) ;
```

3.13 ParU_FreeNumeric: free a numeric factorization

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

3.14 ParU_FreeSymbolic: free a symbolic analysis

```
ParU_Info ParU_FreeSymbolic
```

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

3.15 ParU_FreeControl: free a Control object

```
ParU_Info ParU_FreeControl
(
    // input/output:
    ParU_Control *Control_handle    // Control object to free
) ;
```

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_Control: parameters that control ParU

Any C ParU function can be passed a NULL pointer for its `Control` parameter. In that case, defaults are used. To use non-default parameters, create a `Control` object and then set its parameters. The object is freed by `ParU_C_FreeControl`.

The `ParU_C_Control` structure contains parameters that control various ParU options. The object is created by `ParU_C_InitControl`, modified by `ParU_C_Set_Control_*`, and deleted by `ParU_C_FreeControl`. Its contents can be queried with `ParU_C_Get_Control_INT64`. and `ParU_C_Get_Control_FP64`.

Any ParU function can be passed a NULL pointer for its `Control` parameter. In that case, defaults are used. To use non-default parameters, create a `Control` object and then set its parameters.

```
ParU_Info ParU_C_InitControl
(
    ParU_C_Control *Control_C_handle    // Control object to create
) ;

ParU_Info ParU_C_FreeControl
(
    ParU_C_Control *Control_handle_C    // Control object to free
) ;
```

4.3 ParU_C_Get_*: get information from a ParU opaque object

The `ParU_C_Get_*` methods retrieve scalars or arrays from the C versions of the Control, Symbolic, or Numeric objects. See Section 3.4 for details.

```
ParU_Info ParU_C_Get_INT64       // get int64_t contents of Sym_C and Num_C
(
    // input:
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    ParU_Get_enum field,         // field to get
    // output:
    int64_t *result,             // int64_t result: a scalar or an array
    // control:
    ParU_C_Control Control_C
) ;

ParU_Info ParU_C_Get_FP64        // get double contents of Sym_C and Num_C
(
    // input:
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const ParU_C_Numeric Num_C,  // numeric factorization from ParU_C_Factorize
    ParU_Get_enum field,         // field to get
    // output:
    double *result,              // double result: a scalar or an array
    // control:
    ParU_C_Control Control_C
) ;

ParU_Info ParU_C_Get_Control_INT64  // get int64_t contents of Control
(
    // input:
    ParU_Control_enum field,      // field to get
    // output:
    int64_t *result,              // int64_t result: a scalar or an array
    // control:
    ParU_C_Control Control_C
) ;

ParU_Info ParU_C_Get_Control_FP64   // get double contents of Control
(
    // input:
    ParU_Control_enum field,      // field to get
    // output:
    double *result,               // int64_t result: a scalar or an array
    // control:
    ParU_C_Control Control_C
) ;

ParU_Info ParU_C_Get_Control_CONSTCHAR   // get string from Control
(
    // input:
    ParU_Control_enum field,      // field to get
```

```
    // output:
    const char **result,          // string result
    // control:
    ParU_C_Control Control_C
) ;
```

4.4 ParU_C_Set_Control_*: set Control parameters

The `ParU_C_Set_Control_*` methods set parameters in the C version of the Control object. See Section 3.2 for details.

```
ParU_Info ParU_C_Set_Control_INT64      // set int64_t parameter in Control
(
    // input
    ParU_Control_enum field,    // field to set
    int64_t c,                  // value to set it to
    // control:
    ParU_C_Control Control_C
) ;

ParU_Info ParU_C_Set_Control_FP64       // set double parameter in Control
(
    // input
    ParU_Control_enum field,    // field to set
    double c,                   // value to set it to
    // control:
    ParU_C_Control Control_C
) ;
```

4.5 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
) ;
```

4.6 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
    const 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
) ;
```

4.7 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_C_Factorize`, and the `Sym_C` and `Num_C` structures from that call must be passed to this method.

```
ParU_Info ParU_C_Solve_Axx       // solve Ax=b, overwriting b with solution x
(
    // input:
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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        // solve Ax=b
(
    // input:
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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       // solve AX=B, overwriting B with solution X
(
    // input
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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       // solve AX=B, overwriting B with solution X
(
    // input
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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
) ;
```

4.8 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_C_Factorize`. No scaling or permutations are used.

```
ParU_Info ParU_C_Solve_Lxx       // solve Lx=b, overwriting b with solution x
(
    // input:
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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       // solve LX=B, overwriting B with solution X
(
    // input
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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
) ;
```

4.9 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       // solve Ux=b, overwriting b with solution x
(
    // input:
```

```
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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       // solve UX=B, overwriting B with solution X
(
    // input
    const ParU_C_Symbolic Sym_C, // symbolic analysis from ParU_C_Analyze
    const 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
) ;
```

4.10 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 3.10 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
) ;
```

4.11 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 3.11 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
) ;
```

4.12 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
) ;
```

4.13 ParU_C_FreeNumeric: free a numeric factorization

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

4.14 ParU_C_FreeSymbolic: free a symbolic analysis structure

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

4.15 ParU_C_FreeControl: free a Control object

```
ParU_Info ParU_C_FreeControl
(
    ParU_C_Control *Control_handle_C    // Control object to free
) ;
```

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

6.1 Compiling ParU for MATLAB

To use ParU in MATLAB, you must first compile the `paru` mexFunction. In MATLAB, go to the `ParU/MATLAB` directory and type `paru_make`. Then add the `ParU/MATLAB` directory to your MATLAB path for future use.

For best performance, the `paru` mexFunction relies on functions unique to the Intel MKL BLAS. An optional input, `paru_make(try_intel)`, is true by default. `paru_make` detects the BLAS library used by MATLAB and then attempts to use functions unique to the Intel MKL BLAS library (`mkl*set_num_threads_local`). This may fail when `paru` is compiled, in which case compilation is reattempted with `try_intel` false). If `paru` fails when it runs, with a link error reporting that an an `mkl_*` routine is not found, use `paru_make(false)` to disable the Intel MKL functions.

Limitations: The built-in compiler used by the MATLAB `mex` command on Windows does not support OpenMP, so only parallelism within the BLAS can be used on Windows when using MATLAB.

6.2 Using ParU in MATLAB

The basic usage `x=paru(A,b)` solves the linear system $Ax = b$, computing `x=A\b`. The matrix `A` must be sparse, square, non-singular, and real.

Additional options are available to change `paru`'s behavior, and an additonal output parameter reports statistics on the algorithm:

```
[x,stats] = paru (A,b,opts)
```

`opts` is a struct containing the following fields. Any field that is not recognized is ignored, and missing fields are treated as their defaults:

- `opts.strategy`: ordering strategy, as a string (default: `'auto'`):
  - `'auto'`: the strategy is selected automatically.
  - `'symmetric'`: ordering of `A+A'`, with preference for diagonal pivoting. Works well for matrices with mostly symmetric nonzero pattern.
  - `'unsymmetric'`: ordering `A'*A`, with no preference for diagonal pivoting. Works well for matrices with unsymmetric nonzero pattern.
- `opts.tol`: relative pivot tolerance for off-diagonal entries (default: 0.1). Pivot entries must be 0.1 times the max absolute value in its column.

- `opts.diagtol`: relative pivot tolerance for diagonal pivot entries when using the symmetric strategy (default: 0.001). A lower tolerance for diagonal entries tends to reduce fill-in.
- `opts.ordering`: fill-reducing ordering option, as a string (default: `'amd'`):
  - `'amd'`: AMD for the symmetric strategy, COLAMD for unsymmetric.
  - `'cholmod'`: use CHOLMOD's ordering strategy: try AMD or COLAMD, and then try METIS if the fill-in from AMD/COLAMD is high; then selects the best ordering found.
  - `'metis'`: METIS on `A+A'` for symmetric strategy, `A'*A` for the unsymmetric strategy.
  - `'metis_guard'`: use the `'metis'` ordering unless the matrix has one or more rows with $3.2\sqrt{n}$ or more entries, in which case use `'amd'`.
  - `'none'`: no fill-reducing ordering.
- `opts.prescale`: prescaling the input matrix (default `'max'`).
  - `'none'`: no prescaling.
  - `'sum'`: scale each row by the sum of the absolute values of the row. The prescaled matrix is RA where `R(i,i) = 1/sum(abs(A(i,:)))`.
  - `'max'`: scale each row by the sum of the absolute values of the row. The prescaled matrix is RA where `R(i,i) = 1/max(abs(A(i,:)))`.

`stats` is an optional output that provides information on the ParU analysis and factorization of the matrix:

- `stats.analysis_time`: symbolic analysis time in seconds.
- `stats.factorization_time`: numeric factorization time in seconds.
- `stats.solve_time`: forward/backward solve time in seconds.
- `stats.strategy_used`: symmetric or unsymmetric.
- `stats.ordering_used`: `amd(A+A')`, `colamd(A)`, `metis(A+A')`, `metis(A'*A)`, or none.
- `stats.flops`: flop count for LU factorization.
- `stats.lnz`: number of entries in L.
- `stats.unz`: number of entries in U.
- `stats.rcond`: rough estimate of the recripical of the condition number.
- `stats.blas`: BLAS library used, as a string.

- `stats.front_tree_tasking`: a string stating how the paru mexFunction was compiled, whether or not tasking is available for factorizing multiple fronts at the same time (`'sequential'` or `'parallel'`). Parallel tasking is required for best performance, and requires OpenMP tasking, which is available in OpenMP 4.0 and later.

7 Requirements and Availability

ParU requires several Collected Algorithms of the ACM: CHOLMOD [5, 8], AMD [1, 2], COLAMD [6, 7] and UMFPACK [9] for its ordering/analysis phase and for its basic sparse matrix data structure, and the BLAS [10] 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 [11]. or vendor-provide BLAS. ParU relies heavily on nested parallelism, by making parallel calls to the BLAS, each of which can themselves be parallel. This is supported by the Intel MKL BLAS library with an Intel-specific method (`mkl_set_num_threads_local`) to tell the BLAS how many threads to use. As a result, tor best performance, the Intel MKL BLAS is required.

ParU also relies heavily on OpenMP tasking to factorize multiple frontal matrices at the same time, where each frontal matrix can also be factorized by multiple threads. If tasking is not available, each frontal matrix is factorized one at a time (but still in parallel). For best performance, nested parallelism is required. However, when using the `gcc` compiler on Windows and Mac, we have found that the OpenMP library can hang. As a result, on those platforms, nested parallelism is disabled when using `gcc`. If using `gcc`, use a recent compiler (version 7.5.0 fails; 12.2.0 works).

You can query ParU at run time to determine which BLAS library it is using, and whether or not it is compiled to use parallel or sequential factorization of its frontal matrix tree. See `ParU_Get` (Section 3.4) for details.

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 much 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 memory and time to form it. To avoid such conditions, ParU uses the ordering strategy `PARU_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] Mohsen Mahmoudi Aznaveh. ParU: A task based parallel multifrontal and unsymmetric sparse LU factorization, 2022. PhD thesis, Texas A&M University.

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

[10] 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.

[11] 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.