

PARU: A TASK BASED PARALLEL MULTIFRONTAL AND UNSYMMETRIC SPARSE LU
FACTORIZATION

A Dissertation

by

MOHSEN MAHMOUDI AZNAVEH

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Chair of Committee, Timothy A. Davis
Committee Members, Dezhen Song
Vivek Sarin
Simon Foucart
Head of Department, Scott Schaefer

December 2022

Major Subject: Computer Engineering

Copyright 2022 Mohsen Mahmoudi Aznaveh

ABSTRACT

We are introducing a new sparse direct solver with a parallel multifrontal algorithm. The algorithm is designed by borrowing the symbolic analysis phase and some ideas from UMFPACK. UMFPACK is a part of the SuiteSparse package and appears as a built-in routine in MATLAB. Although the general idea of the algorithm is the same, we designed a new algorithm from scratch that is more amenable to parallelism. The new algorithm is right-looking multifrontal with rectangular fronts and uses the sparsest pivot like UMFPACK; however, its data structures are different, and the factorization is more coarse-grained. One of the most significant contributions of this work is the algorithm that cuts various dependencies of the data structure. The only dependency comes from the matrix's pattern from the elimination tree.

Independent tasks can start working in different computing cores using OpenMP tasking. Each task can call BLAS kernels, specifically matrix multiplication and triangular solve. Therefore, there is nested parallelism in this algorithm. To better manage hardware resources, we can exploit parallel BLAS only if we have more computing cores than tasks. In practice, the performance depends on the BLAS library and the input matrix. It is better to have a mixed strategy to have both parallel BLAS with parallel fronts. Therefore, our algorithm is less affected by parallel BLAS and shows good performance compared to UMFPACK. Data structure and memory management are different in ParU, so it needs less memory to solve a system.

The current algorithm is implemented for a shared memory environment. However, the algorithm can be an excellent candidate to be implemented in a distributed environment. Moreover, while there are parallel BLAS calls (typically larger than what UMFPACK has), it is also an excellent candidate for using hardware accelerators like GPUs.

DEDICATION

To my mother and my father

ACKNOWLEDGMENTS

I would like to thank Dr. Tim Davis for his constant support. This research would never happen without his guidance. I would also like to thank my doctoral research committee: Dr. Dezheng Song, Dr. Vivek Sarin, and Dr. Simon Foucart. I am grateful to Dr. Wissam Sid-Lakhdar for his help, especially for getting the results from MUMPS and SuperLU-MT. And to all the friends I made along this long journey, Dr. Scott Kolodziej and Dr. Jinhao Chen. And thank you to Dr. Ahmad Mahmoudi Aznavah, my brother, whose support carried me this far.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Timothy A. Davis, [advisor] of the Department of Computer Science and Engineering, and committee members, Professors Dezhen Sources and Vivek Sarin from the Department of Computer Science and Engineering and Professor Simon Foucart from the Department of Mathematics.

All the work in this dissertation is completed independently, and the code base is open source and is distributed under the GNU GPL license.

Funding Sources

The graduate study was supported by the following funding sources:

- Research startup funding for Dr. Timothy Davis through the Department of Computer Science and Engineering at Texas A&M University.
- A Graduate Teaching Fellowship from the College of Engineering
- Corporate gift funding from Intel, Nvidia, Redis, MathWorks, and Julia Computing

NOMENCLATURE

AMD	Approximate Minimum Degree
API	Application Programming Interface
COLAMD	Column Approximate Minimum Degree
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
Etree	elimination tree
FLOPS	floating point operations per second
GPU	Graphics Processing Unit
I	Identity matrix
MKL	(Intel) Math Kernel Library
OpenMP	Open Multi-Processing
ParU	a PARallel Unsymmetric-pattern multifrontal package
UMFPACK	an Unsymmetric-pattern MultiFrontal PACKage

TABLE OF CONTENTS

	Page
ABSTRACT	ii
DEDICATION	iii
ACKNOWLEDGMENTS	iv
CONTRIBUTORS AND FUNDING SOURCES	v
NOMENCLATURE	vi
TABLE OF CONTENTS	vii
LIST OF FIGURES	ix
LIST OF TABLES.....	xi
1. INTRODUCTION AND LITERATURE REVIEW	1
1.1 Introduction.....	1
1.2 Mathematical background	1
1.2.1 BLAS	2
1.2.2 Sparse solvers.....	2
1.2.2.1 Different flavors of LU factorization	3
1.3 Background on frontal and multifrontal methods	5
1.4 Fill-reducing ordering.....	7
1.4.1 Multifrontal methods	7
1.4.2 UMFPACK	9
1.4.3 Parallel methods	10
1.4.4 OpenMP and tasking	11
2. PARU ALGORITHM	12
2.1 Introduction to ParU	12
2.2 Symbolic Analysis	12
2.3 Numerical Factorization	16
2.3.1 Single Front assembly	18
2.3.2 Pivotal column assembly	20
2.3.3 Pivotal column factorization	21
2.3.4 U-part assembly and update.....	24
2.3.5 Contribution block assembly	25

2.3.5.1	ParU prior front assembly	28
2.3.5.2	Finalizing the heap	29
2.3.6	Summary of Numerical Factorization algorithm	30
2.4	Solve	31
2.5	Parallelism	32
2.5.1	scheduling.....	37
2.5.1.1	BLAS scheduling.....	39
2.5.2	Other Parallelism Options.....	40
2.6	Performance comparison with UMFPACK	41
3.	PARU CODE STRUCTURE AND RESULTS	44
3.1	C/C++ Example	44
3.1.1	C/C++ Syntax.....	46
3.1.2	Details of the C/C++ Syntax	47
3.2	Requirements and Availability	48
3.3	Code Anatomy	48
3.4	Experimental Results	50
4.	SUMMARY AND CONCLUSIONS	57
	REFERENCES	58
	APPENDIX A. DETAILED PERFORMANCE METRICS AND COMPUTING ENVIRON- MENT.....	62

LIST OF FIGURES

FIGURE	Page
1.1 $A = L \times U$	1
1.2 Right-looking LU	4
1.3 Up-looking LU	4
1.4 Left-looking LU	5
1.5 An example of two fronts of a matrix	6
2.1 The original Matrix $b1_ss$	13
2.2 After column pre-ordering and the staircase structure	14
2.3 column elimination tree	14
2.4 Singleton structure	15
2.5 renumbered tree	16
2.6 Augmented tree. Boxes refer to (renumbered) original rows of the staircase matrix S , and circles refer to frontal matrices.	17
2.7 Finding rows of an active front	20
2.8 Assembly of pivot columns	21
2.9 Factorize the pivot columns	22
2.10 Panel factorization	23
2.11 Assembling U part numerical values	24
2.12 Applying $-l_{21} \times u_{12}$	25
2.13 The prior contribution block is within the current front	26
2.14 Prior contribution block rows are a subset of current front rows	27
2.15 Prior contribution block columns are a subset of current front columns	27

2.16	Prior contribution block rows and columns have rows and columns in common with the current front (No assembly)	28
2.17	ParU factors relative to the input matrix	33
2.18	Elimination tree of the input matrix	34
2.19	Task tree of matrix from Figure 2.18	35
2.20	Ready fronts to be executed	35
2.21	The tree in the middle of the execution	36
2.22	An example of how the task can partition the tree	38
3.1	ParU_Factorize	49
A.1	Intel Vtune graph for ParU	63
A.2	Topology of our machine	64

LIST OF TABLES

TABLE	Page
2.1 time consuming functions in ParU	38
3.1 ParU user-controllable variables.....	47
3.2 List of matrices	52
3.3 Comparison between different methods (total time for analysis, factorization, and solve, in seconds)	53
3.4 ParU and UMFPACK detailed comparison using AMD or COLAMD (run time in seconds).....	55
3.5 ParU and UMFPACK detailed comparison using METIS (run time in seconds)	56

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Introduction

ParU is a C++ package to find the direct solution of systems of linear equations, $Ax = b$ where A is a sparse, square, and unsymmetric matrix. This package uses OpenMP tasking for finding the solution in parallel. The matrix PAQ is factorized into the product LU , where P and Q are permutation matrices that are chosen to preserve sparsity. ParU is using a multifrontal approach like [1, 2, 3, 4, 5]; the most important aspect of ParU is that it is developed to be able to run different fronts in parallel. The only restriction for running the fronts come from the elimination tree (*etree*). Multifrontal methods compute the factors using dense matrix kernels such as BLAS in each front. The use of dense BLAS kernels can lead to a nice performance compared to other methods. In multifrontal methods, BLAS kernels are typically larger than supernodal methods and can exploit BLAS level 3 more.

1.2 Mathematical background

Factorization, in mathematics, is the process of decomposition of mathematical objects into the product of other objects, known as factors. The matrix factorization decomposition techniques used in the numerical analysis include Cholesky, LU, and QR, to mention a few. The solution to linear systems lies in the heart of numerical factorization. Solving systems of linear equations of the form $Ax = b$ needs the factorization algorithm to decompose the matrix into an upper and lower triangular matrix (Figure 1.1).

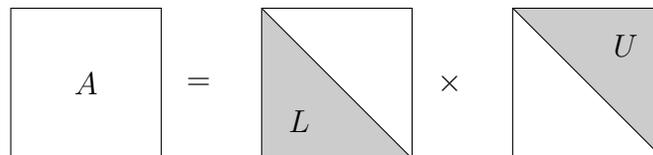

$$A = L \times U$$

Figure 1.1: $A = L \times U$

After computing the factors, solving $Ax = b$ contains just triangular solves. We need to solve $LUx = b$, let $Ux = y$, then $Ly = b$ and $Ux = y$. First, solve $Ly = b$ for y (*forward elimination*) and then solve $Ux = y$ for x (*backward substitution*). LU decomposition, a matrix form of Gaussian elimination with partial pivoting, is the most widely used algorithm for solving linear systems.

1.2.1 BLAS

BLAS is the de facto industry standard for low-level dense linear algebra and is typically tuned for a certain machine's performance. Most mathematical software uses BLAS-compatible libraries like LAPACK [6].

In ParU, whenever possible, we use BLAS kernels. There are implementations for BLAS that take advantage of SIMD operations and has been optimized for different accelerators [7]. We use BLAS-compatible libraries for our dense kernels whenever possible. There are three sets of routines, called levels, for BLAS functionality, which is essential to know when you are working with BLAS.

Level 1 includes all the routines that do vector operations, like vector addition (SAXPY). Level 2 consists of matrix-vector operations, such as matrix-vector multiplication (GEMV). Level 3 contains matrix-matrix operation, for example matrix-matrix multiplication (GEMM). The ratio of floating-point operations (n^3) per input size (n^2) for BLAS level 3 is n , and it is more efficient in terms of cache usage and is widely used in sparse linear solvers [2].

1.2.2 Sparse solvers

Sparse matrices have enough zero entries to be taken advantage of [8]. Sparse matrices are found in many areas of engineering and have many applications. Both direct and iterative methods are used to deal with sparse matrices. Iterative methods generate a sequence of $x^{(k)}$, which converges to the solution of $Ax = b$. Improving direct solvers can also help iterative methods for finding better-predefined parameters. Generally, iterative solvers need less disk and memory space than direct solvers, but their results strongly depend on preconditioning. Iterative methods are out

of the scope of this dissertation.

In sparse algorithms, we do not store zero entries explicitly. A simple data structure would be keeping non-zero elements in an arbitrary form called *a triplet*. This format is easy to generate but hard to use and, most of the time, not cache-friendly. An alternative way is to store non-zero entries column/row-wise, in a format called *column/row compressed*. By avoiding unnecessary computation on zeros, the amount of work that needs to be done is much less.

Although sparse algorithms share many fundamentals with dense (or not-sparse) counterparts, they need different methods and careful designs to be efficient. In sparse solvers, there are usually three steps to solve the system $Ax = b$. The first phase is **symbolic analysis**, which involves examining the matrix pattern and determining dependencies. ParU and UMFPACK do not consider numerical values in this phase, but there are methods that do pre-scaling and permuting to put large entries on the diagonal. ParU and UMFPACK scale rows of the matrix by default but not in the symbolic analysis phase. This makes it easier to separate symbolic analysis and do this phase once for the matrices with the same pattern.

The second step is **factorizing** the matrix and computing the numerical values of L and U . The last step is **solving**, forward elimination, and backward substitution (two triangular solves).

1.2.2.1 Different flavors of LU factorization

There are many different sparse factorization algorithms implemented, each for optimizing one or several special characteristics. Different economies are also possible for matrices with special properties, and they might have different applications.

A right-looking algorithm factorizes the matrix from top-left to bottom right (Figure 1.2), and as it progresses, it modifies the submatrix to the right. In Figures 1.2, 1.3 and 1.4 the gray part is the focus of the algorithm. In Figure 1.2, the blue part resembles the part of the original matrix that needs to be modified. Other variants do not change the original matrix. The right-looking method is a head recursion algorithm, and by applying the Schur complement of A_{22} block, we have a smaller matrix to factorize. The right-looking LU is harder to implement in general, and we have to use a pivoting strategy for numerical stability.

$$\begin{array}{|c|c|} \hline l_{11} & 0 \\ \hline l_{21} & L_{22} \\ \hline \end{array} \times \begin{array}{|c|c|} \hline u_{11} & u_{12}^T \\ \hline 0 & U_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline a_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \\ \hline \end{array}$$

Figure 1.2: Right-looking LU

The up-looking LU (Figure 1.3) is a tail recursion algorithm in comparison, and by unrolling the recursion, the algorithm also computes L and U from top left to bottom right. However, the data movement is very different than the right-looking. In this method, there can be no pivoting based on the numerical values of the pivot row or column, since those have yet to be computed.

$$\begin{array}{|c|c|} \hline L_{11} & 0 \\ \hline l_{21}^T & l_{22} \\ \hline \end{array} \times \begin{array}{|c|c|} \hline U_{11} & u_{12} \\ \hline 0 & u_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_{11} & a_{12} \\ \hline a_{21}^T & a_{22} \\ \hline \end{array}$$

Figure 1.3: Up-looking LU

Left-looking is another variant of LU algorithms that computes columns from left to right, one column (or supernode) at a time (Figure 1.4). It is referred to as left-looking, since at each stage, it computes the current pivot column by using the factors of L and U to the left of the current pivot column. The left-looking method does not modify the active submatrix to the right of the current pivot column. Both left-looking and right-looking algorithms can be used for pivoting. However, the right-looking method in sparse cases had the advantage of finding a sparse pivot row while the pattern of $A^{[k]}$ is in hand.

$$\begin{array}{|c|c|c|} \hline L_{11} & & \\ \hline l_{21}^T & 1 & \\ \hline L_{31} & l_{32} & L_{33} \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline U_{11} & u_{12} & U_{13} \\ \hline & u_{22} & u_{23}^T \\ \hline & & U_{33} \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline A_{11} & a_{12} & A_{13} \\ \hline a_{21}^T & a_{22} & a_{23}^T \\ \hline A_{31} & a_{32} & A_{33} \\ \hline \end{array}$$

Figure 1.4: Left-looking LU

Matrix factorization frequently has columns and rows with duplicate structures. The *supernodal* methods are the methods that exploit this property to save space by storing fewer integers and time by doing dense matrix operations. Most modern sparse solvers use this property in some way. SuperLU [9] is a left-looking method introduced by Demmel, Eisenstat, Gilbert, Li, and Liu. Demmel, Gilbert, and Li [10] introduce a parallel shared-memory version of SuperLU. We will discuss parallel solvers in section 1.4.3.

There is a survey by Davis, Rajamanickam, and Sid-Lakhdar on direct methods and sparse linear systems [11] and the history of different packages. Here we explain the methods that are most relevant to our implementation. For more information on other solvers, please refer to the survey [11].

1.3 Background on frontal and multifrontal methods

Frontal methods were first introduced by B. M. Irons [12]. The method was initially developed to solve symmetric positive-definite banded linear systems, but Hood [13] generalized the method for unsymmetric cases. The stiffness matrix is represented as the sum of finite-element contributions in the finite-element method. Each element is only associated with a smaller set of variables. A front, which is a dense submatrix, may be eliminated before fully assembled. The variables that are being eliminated need to be fully-summed; there is no more contribution to that column or row. However, other variables do not need to be fully-summed, and summing variables can be done in any order.

The frontal system for an assembled system can be written as

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix} \quad (1.1)$$

F_{11} is fully-summed. Therefore, pivots can be chosen from the first block column, and numerical pivoting can be done while the rows are fully-summed. The Schur complement can be computed using dense matrix computation:

$$F_{22} - F_{21}F_{11}^{-1}F_{12} \quad (1.2)$$

In a frontal method, first, a *front* is defined and allocated. Then the method assembles fronts and eliminates and updates variables; the elements are assembled, and a partial factorization is applied on the front when fully assembled. The fully-summed variables are eliminated, and other variables get updated with each elimination. Eliminated variables are no longer needed, and this process continues to eliminate all the variables. The solution of the system can be obtained with forward elimination and backward substitution.

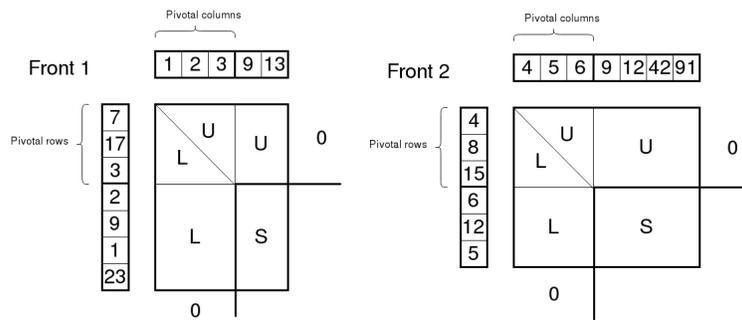


Figure 1.5: An example of two fronts of a matrix

Figure 1.5 is an example that shows two arbitrary fronts of a matrix. L and U parts of the front are fully assembled and updated and can be stored in the disk, and S is the Schur complement that

will be saved after the computation and assembled in future fronts.

1.4 Fill-reducing ordering

Finding a permutation P with fewer non-zeros in its factorization $LU = PA$ is an NP-hard problem [14]. Therefore, there are heuristics for solving this problem. The greedy algorithm *minimum degree* chooses the sparsest pivot row and column. The pivot must also be numerically large enough compared to the absolute maximum value in the pivot column. For the symmetric case, a permutation P must be found so that PAP^T has a Cholesky LL^T or LDL^T factorization. The minimum degree problem has several different variants for different types of problems.

Computing the exact degree of nodes is costly, and there is an algorithm to compute the degree approximately (AMD) [15, 16]. The idea of approximation was first derived from rectangular fronts in UMFPACK [3], which is an unsymmetric algorithm. AMD computes an upper bound for the degree of each node in Cholesky computation. Using the upper bound is faster and requires less computation than computing the accurate degree. Moreover, the minimum degree algorithm is a heuristic per se, and in practice AMD finds an ordering of equal quality in much less time, as compared to using the exact degree. COLAMD [17] is a variant of the approximate minimum degree algorithm that computes an ordering of $A^T A$ without explicitly forming it.

Nested dissection, which was introduced by Birkhoff and George [18] is also a fill-reducing ordering that works better on matrices arising from discretizations of two- or three-dimensional problems. Like the AMD algorithm, the goal of the nested dissection algorithm is to reduce fill-in. By employing nested dissection algorithms, a *vertex separator* is found that divides the graph into two roughly equal-sized subgraphs. After removing the nodes, the subgraphs are recursively ordered. This sort of pre-orderings is more suitable for having a bushy elimination tree and, therefore, more parallelism in the tree. METIS [19, 20] is a known example of this sort of algorithm.

1.4.1 Multifrontal methods

Duff and Reid [21] generalized the frontal method by Irons [12] and developed the multifrontal method. The multifrontal method was developed to perform LL^T or LDL^T factorization of a

symmetric matrix, however, it can offer a general framework for solving unsymmetric systems, and it can also be a basis for a sparse QR factorization.

One of the first detailed descriptions of the multifrontal method is given by Liu [5]. The finite-element formulation of the frontal method is

$$A = \sum_i A^{(i)} \quad (1.3)$$

In which $A^{(i)}$ is the contribution of a finite element. The frontal method is the sequential bracketing of the equation 1.3. The multifrontal method can be interpreted as a generalization to any bracketing, and the *assembly tree* can be interpreted as the expression of this bracketing. There are three phases in a multifrontal method: *analysis phase*, *numerical factorization* and *solve phase*.

In the analysis phase, the elimination tree is computed. Each node in the elimination tree represents a dense matrix (or front) much smaller than the sparse input matrix. The factorization of the sparse matrix is then partial factorization of the fronts. The assembly tree is the supernodal version of the elimination tree. However, the term elimination tree is used for both trees in literature.

The tree is traversed from the bottom to the top in the numerical factorization phase. The only theoretical constraint is that the computation of a front must be done after the computation of all its children. Like the frontal method, partial factorization is applied to fully-summed rows and columns, then the *contribution block* is updated (Schur complement). In the multifrontal method, the assembly of the contribution blocks of all the children happens in the parent, while in the frontal method, only one child is assembled.

In the solve phase, forward elimination is applied from the bottom to the top, and backward substitution is applied from top to bottom of the tree. A dense triangular solve is needed for each front to compute the solution of the system. We can have tree-level parallelism both in the factorization and solve phases. While the factorization phase is the most time-consuming part of solving the system, we have only implemented tree-level parallelism in the factorization phase of ParU.

MUMPS (MULTifrontal Massively Parallel sparse direct Solver) [22] is a symmetric-pattern multifrontal solver. Although it can solve unsymmetric-pattern matrices, it has to use symmetric analysis for them. Symmetric-pattern multifrontal algorithms are typically easier to implement. We discuss the reason in section 2.3.5. Davis and Duff developed the original unsymmetric-pattern multifrontal technique, UMFPACK [3]. The method keeps track of the approximate degrees of the current submatrix's rows and columns, and a symmetric form was later incorporated into AMD's minimal degree ordering algorithm.

1.4.2 UMFPACK

UMFPACK [23, 3] is a right-looking LU factorization with dynamic pivoting. It is used inside MATLAB for solving $x=A \setminus b$ when A is unsymmetric. UMFPACK can be found in the SuiteSparse package, and it can solve a general problem $Ax = b$ where A is sparse and unsymmetric and the matrix PAQ or $PRAQ$ is factorized into LU . Q is the column ordering which is chosen to have a priori upper bound on fill-in, and it changes during factorization. P is the row order and is mainly selected for numerical stability during factorization. R is the diagonal matrix for scaling and scales the rows of A . Note that UMFPACK can solve pattern symmetric and numerical symmetric systems, but the algorithm is not designed to exploit symmetry, so it is not as efficient as symmetric algorithms.

The algorithm is computing each front sequentially and is designed very carefully to take advantage of a single processor. However, it has the powerful ability to use parallel BLAS for dense submatrix computation. The algorithm is fairly complex; our goal in this part is to give an overview of the algorithm. We have to mention some implementation details, especially where the implementation differs from ParU.

In UMFPACK, pivots are chosen one by one, and the active front is kept as small as possible in each stage. A pivot is chosen based on numerical value and sparsity of both column and row. After choosing the pivot, the active front is expanded if necessary, and the degrees of rows and columns corresponding to that node are updated. In addition to the complex design of data structures for storing the active front and results, UMFPACK also has its own garbage collec-

tion method that, whenever necessary, shrinks memory to better use the available space during the runtime. UMFPACK manages the memory itself, and the number of memory allocations is approximately constant, independent of the size of the problem.

In the analysis phase, UMFPACK finds the row ordering and the column pre-ordering, then both the row and column orderings are revised during numerical factorization to preserve numerical stability. In UMFPACK, there is just one active front at any stage of the algorithm. This front is assembled from prior fronts and elements. A list of *tuples* for each row and column keeps track of elements on that row/column and contains the front number and relative row/column number.

These lists are typically short and are intended to be kept as short as possible by assembling prior fronts and trimming the list at the first available opportunity. Therefore, scanning tuple lists are generally fast; by scanning these lists, UMFPACK does operations like set union and set intersection. These tuples are used for finding the set union of fully assembled columns as well as for updating the degrees of rows and columns.

1.4.3 Parallel methods

Parallel methods have a long history in sparse solvers. Parallelism first appeared in Calahan's work [24], and since then, there have been some parallel implementations of the most famous packages. The parallel frameworks also have changed drastically during these years. The technologies have improved, and new hardware and software have been introduced. There are different methods of exploiting distributed environments and also shared memory environments. Here we try to discuss the most relevant methods. SuperLU-MT [9] is the shared memory version of the left-looking algorithm SuperLU. SuperLU has a right-looking variant, SuperLU-DIST [25] which is a distributed-memory implementation.

Parallelism and exploiting dense submatrices are different ways modern sparse direct solvers perform. UMFPACK [3], as an example, does not use parallelism between fronts, but it uses the dense matrix computation. Duff [26] surveyed techniques to make direct solvers more efficient. The implementation of dense matrix operations, which significantly impact the performance of sparse direct solvers, is beyond the scope of this dissertation ([27, 6, 28, 29]).

MUMPS [22] is a multifrontal solver with many different variations. It can exploit shared memory parallelism, and the design is for symmetric-pattern matrices. For the shared memory version, MUMPS cuts the etree, solves different subtrees in parallel, and finally solves the subtree containing the root. For a complete list of parallel direct solvers, please refer to the survey by Davis, Rajamanickam, and Sid-Lakhdar [11].

1.4.4 OpenMP and tasking

OpenMP is an application programming interface (API) for shared-memory multiprocess programming and is supported in C, C++, and Fortran language. OpenMP follows a fork-join model, and the threads run concurrently. Each thread is assigned the resources it needs. OpenMP is designed to be simple to use, and it is a portable API. The main philosophy of OpenMP is to get the sequential code and, with minimal effort, change it to a parallel code with reasonable efficiency.

Up to version 3.0, the primary way of parallelizing OpenMP was exploiting regular loops and ignoring irregular parallelism. Tasking added in version 3.0 in May 2008 [30] allows OpenMP to exploit parallelism in irregular problems. Especially in the area of sparse matrix algorithms that deal with jobs with arbitrary sizes, tasking is necessary.

In our multifrontal algorithm, tasking can be easily mapped to the execution of fronts. In OpenMP tasking, when a thread encounters a task construct, it may choose to execute the task immediately or defer the task. Ideally, a pool of tasks can be executed with variant sizes. The only thing we needed to figure out in ParU was how to enforce the constraint of the tree in OpenMP.

One of the downsides of using OpenMP is that Microsoft only supports OpenMP 2.0. Also, the programmer has little control over how and when the tasks are executed.

2. PARU ALGORITHM

2.1 Introduction to ParU

The main purpose of this project is to design and implement an unsymmetric direct sparse solver like UMFPACK in which there is more than one active front. UMFPACK is designed as a sequential algorithm and can deal with only one front at a time. UMFPACK uses $\mathcal{O}(n)$ algorithms to union set or intersection set for both rows and columns. For these types of algorithms, you need to have a gather-scatter data structure for storing the indices. The main challenge to making UMFPACK parallel comes from managing the columns. Each front owns rows, and each thread can safely manage them within a single gather-scatter space. For a parallel algorithm that works like UMFPACK, we need a version that can work on different columns at a time which either needs several gather-scatter spaces or a way to manage a single data structure in parallel. We decided to implement ParU to cut the dependencies of columns by using general set algorithms, like the red-black tree algorithm, typically in $\mathcal{O}(n \log n)$.

Like most other solvers, we solve the problem in three phases: 1) symbolic analysis, 2) numerical factorization, and 3) solve. The next sections look closely at how these different parts are implemented. In the implementation of ParU, there is not much parallelism during symbolic factorization. The most important parallelism is during numerical factorization among fronts, and we have used BLAS kernels during solve phase.

2.2 Symbolic Analysis

After reading the sparse input matrix, the algorithm's first step is doing the symbolic analysis. The symbolic analysis phase probes only the pattern of the matrix and extracts useful information for the factorization. For reading the data, we use CHOLMOD reading functions inside Suite-Sparse. Then we call UMFPACK's symbolic analysis computes the elimination tree and has an upper bound for the sizes of the fronts.

After that, ParU makes the tree more compact in the relaxed amalgamation phase. Basically,

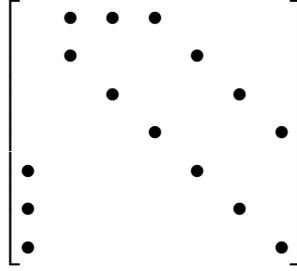


Figure 2.1: The original Matrix b1_ss

a child and its parent merge if their count of pivotal columns is less than some threshold. The default is 32, and the user can change it using the `Control` options in ParU. This phase can help to reduce small fronts. The depth of each front in the elimination tree is also computed in the symbolic phase. We will explain how the depth of the front can be useful in our parallel algorithm.

UMFPACK gives ParU a column pre-ordering, and ParU changes the row ordering so that nonzeros in the columns be closest according to their new column index. Making this row ordering is called forming the *staircase structure* of the matrix, while this structure looks like a staircase. In the staircase structure, rows of the matrix are sorted in ascending order according to each row's column index of the leftmost nonzero element. The staircase structure of the tree is computed in the symbolic analysis phase in ParU, which generally makes the cache usage more efficient. The pattern of the matrix b1_ss from the SuiteSparse Matrix Collection [31] is shown in Figure 2.1. Figure 2.2 shows this matrix after column pre-ordering done by UMFPACK and pre-ordering of rows to form the staircase structure. The column elimination tree of this matrix is shown in Figure 2.3. Note that the last front has two pivotal columns, and for demonstration purposes, we set the relaxed amalgamation threshold to 1.

UMFPACK, during the symbolic analysis phase, also returns the rows and columns with zero Markowitz cost. In ParU, while computing the staircase data structure, those rows and columns (*singletons*) are identified and saved in a column- or row-based data structure to be used later in solve phase.

ParU uses a different approach to treat singletons. In ParU, the row singletons are kept in a

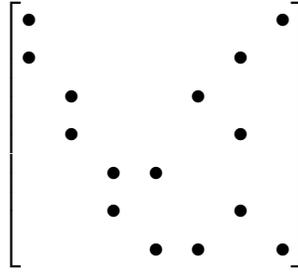


Figure 2.2: After column pre-ordering and the staircase structure

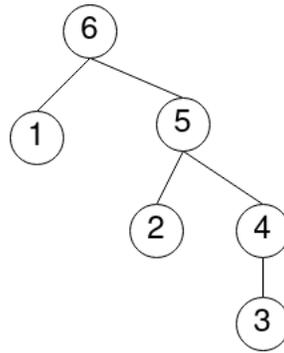


Figure 2.3: column elimination tree

compressed sparse row (CSR), and the column singletons are kept in a compressed sparse column (CSC) data structure. These singletons are not needed during factorization but only during the solve phase. Keeping them in these data structures makes the solve phase easier. Eliminating singletons can affect the symmetry of the matrix in some cases. Therefore, the user has the option not to let eliminate singletons both in UMFPACK and ParU with

```
umf_Control[UMFPACK_SINGLETONS] = 0.
```

Note that the numerical values of singletons are not touched during the symbolic phase. The structure of the singletons is depicted in Figure 2.4. In ParU, U is saved as CSR, and the L part is saved in a CSC format and used during the solve phase.

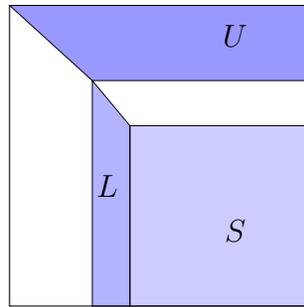


Figure 2.4: Singleton structure

ParU makes a new tree, the *augmented tree*, after the amalgamation phase. The augmented tree is a *row-merge tree* containing rows as elements. A row-merge tree has only original rows on the leaves. The augmented tree is a generalization of that concept into a postordered tree. ParU has to renumber elements and add rows to the data structure to keep it postordered. Using this technique, ParU can start with row elements in the beginning and know which rows belong to which front by looking into an element number. It is also beneficial for parallel implementation, where the dependencies are clear just by looking into the element number within the tree.

The process of making the augmented tree is shown in Figures 2.5 and 2.6. These trees are from the matrix with column elimination tree in Figure 2.3. We have a matrix with seven rows and an elimination tree with six nodes. The augmented tree will have $7 + 6$ nodes, and the leaves will be matrix rows. For this example, element 11 (front 3 in the original elimination tree) has two children 9 and 10, that one is an original row, and the other is a front.

The augmented tree must be postordered. Each node in the augmented tree is an *element*. Each internal element is a front, and each leaf element is an original row. Each element owns the entire descendent rows. Therefore, any active front has a collective interval of rows owned by that front. This makes it more efficient to work with the concept of ownership in the parallel domain. ParU first forms the renumbered tree to make the augmented tree (Figure 2.5), which is the etree but adds the number of the original rows. The augmented tree is shown in Figure 2.6. The leaves are shown in rectangles in this Figure.

In Figure 2.6, front 11 owns all the rows from $[7, 11]$. Generally, a front F always owns the entire interval $[leftmostchild, F-1]$. The leftmost child is also computed in the symbolic analysis. Moreover, by using an augmented tree, we can simplify having more than one active front while there is no data dependency among rows. Note that columns might be shared between two or more active fronts, and we must avoid concurrent data access.

In the symbolic analysis phase, we also compute a task tree. Each task in the task tree includes one or more fronts of the elimination tree. Each chain inside the elimination tree is a single task. Small tasks in the leaves also merge to form bigger tasks. We use OpenMP tasking to run these tasks in parallel.

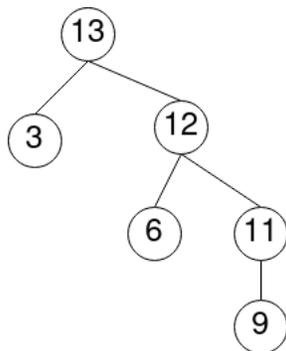


Figure 2.5: renumbered tree

2.3 Numerical Factorization

The numerical factorization computes the values and final pattern of the LU factors using the symbolic analysis and the input matrix. An overview of the algorithm is given in Section 2.3.6. The method iterates over the frontal matrices (in parallel), assembling components of the Schur complement from prior frontal matrices, factorizing the front, and then preparing the Schur complement of the front for assembly into subsequent frontal matrices.

Prior to factorizing any frontal matrices, ParU has to save the numerical values of each row into the element list. Basically, in this phase, each row is saved as a prior front, and after this, ParU

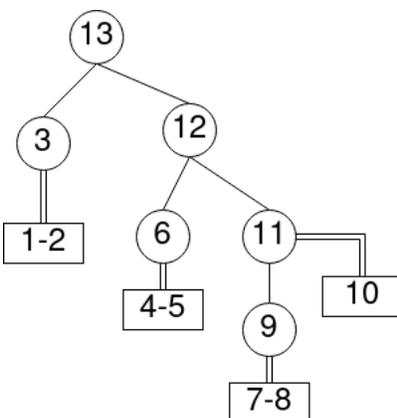


Figure 2.6: Augmented tree. Boxes refer to (renumbered) original rows of the staircase matrix S , and circles refer to frontal matrices.

can start assembling fronts in parallel. The first step of numerical analysis is to form the numerical values staircase structure matrix, S (Figure 2.2), along with the values of singletons. Singleton values will be kept for the solve phase, and there is no need to account for them in the factorization phase. The matrix S is stored by row, which makes forming the row elements simpler. Having the matrix S , ParU can form row elements in parallel and make it ready for the next phase.

In ParU, the only data dependency is from the child to the parent. However, while we are working on an unsymmetric LU factorization method, all the data does not necessarily go strictly from the child to the parent. It can go from a child to a grandparent or further. The data structure to do this and how it is accessed is very important for the solver's performance. UMFPACK uses a set of paired numbers, *tuples*, for both rows and columns to keep track of available rows and columns. ParU also uses tuples for the rows but does not use column tuples. The main reason is that each front owns a range of rows that are computed in symbolic analysis, but they can share columns. Accessing and changing columns' data structures, with multiple owners, in parallel is a challenge.

Sorting the contribution block columns according to the global column index of the matrix is the main idea for getting rid of the column tuples in ParU. The sorting is only done once during the creation of the contribution block. This characteristic can be helpful at several points in ParU.

For each front, ParU additionally keeps track of the index of the first column that hasn't been eliminated or the least active column (*lac*). Note that elimination is done for rows or columns, and keeping track of the front's *lac* or first active column is free. We will explain how we exploit this property during the explanation of the algorithm.

In the rest of this section, we explain the ParU algorithm design, which is suitable for parallelism. In the section 2.5 we will explain how the parallelism is implemented.

2.3.1 Single Front assembly

In ParU, there can be more than one active front. Let's first explain how a single front is assembled and factorized, and then we talk about how it can be parallel. Assembly of the current front is done in three different places: 1) pivotal columns, 2) U-part, and 3) future prior front. Before assembling pivotal columns, we should find the list of the rows contributing to current pivotal columns. In the ParU symbolic analysis phase, pivot columns are identified, and the matrix is permuted to have a staircase structure. Knowing the pivot columns, we can compute the row pattern of the current front. The row pattern is basically a set union of row indices of pivot columns. In Figure 2.7, you can see a scheme of the first steps of making an active front. In this phase, the number of rows of the active front is fixed and only depends on the pivot columns and the matrix structure.

In our earlier implementation, a shared data structure is used to compute the set union of rows, and it is a $\mathcal{O}(n)$ algorithm. We just scanned the pivotal columns' tuples with column tuples and found the set union. In our newer version, we use *lac* to find the list of elements that contribute to the pivotal columns.

A descendant contributes to the pivotal column of the current front if its global column indices lie inside its range of pivotal columns. This property is used in ParU, and the list of pivotal columns is put together and calculated using a heap data structure. A list of live descendant fronts that can contribute to each front is kept; a prior live front is a front with at least a column or row not assembled yet. This list is maintained as a heap, and the key of the heap is *lac*. Finding the columns that go into the pivotal columns is thus as simple as extracting the minimum from the

heap. The algorithm for finding the set of rows is shown in Algorithm 1.

Algorithm 1: Finding the set of rows that appear in the current frontal matrix

Data: Set of live children, $col_1 < pivotal_columns < col_2$

```

for  $1 \rightarrow num_{children}$  do do
  if  $child_i$  is still alive then
    if  $lac(child_i) < col_2$  then
      Add  $child_i$  to the pivotal columns and remove them from the list;
       $set_{rows} \leftarrow set_{rows} \cup rows(child_i)$ ;
    end
  end
end

```

ParU extracts from the heap until the point at which lac is outside the range of the pivotal columns. If the lac is less than the upper bound of the current front's pivotal column, then the prior front contributes to the current pivotal columns. ParU keeps these prior elements in a list and removes them from the heap.

ParU determines the number of rows in this front after assessing the list of elements contributing to the pivotal columns. This number is the set union of all the rows of the contributing items. ParU accomplishes this using an algorithm similar to UMFPACK and a global table that requires $\mathcal{O}(1)$ time per entry in the set of input rows which in total is a $\mathcal{O}(n)$ algorithm. A single table can be utilized for this set union for all of the fronts because of the fact that each active front owns all the rows, and there is no conflict among them.

After finding the number of rows, the amount of memory needed for pivot columns is known, and the memory can be allocated. Then numeric values can be assembled into that memory (the gray part in Figure 2.7). Note that in ParU, the amount of memory required for a frontal matrix is always computed first, then allocated. Therefore, in general, it can use less memory than UMF-PACK, which performs its memory allocation based on an upper bound on the size of the frontal matrix.

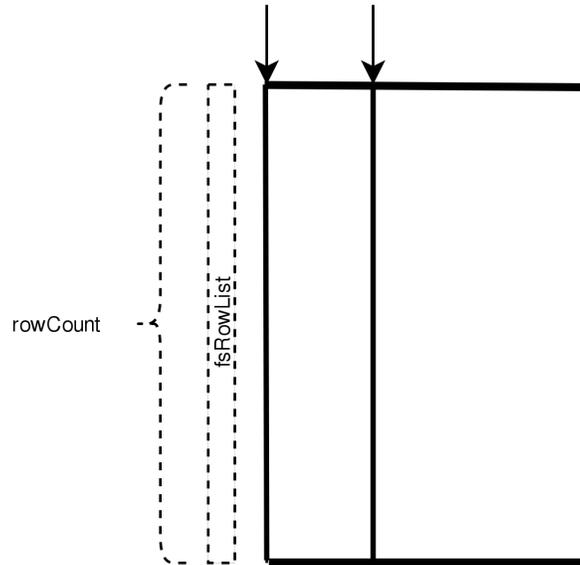


Figure 2.7: Finding rows of an active front

2.3.2 Pivotal column assembly

All the descendants of the current front (the ones that are not eliminated) contribute to the pivotal column of this front if they have a global column within the range of pivotal columns of the current front. As we explained, we use this property, and assembly and computation on the pivotal columns are done using a heap data structure. Prior fronts column indices are sorted based on the global column index of the matrix at the time of generation. In elimination phases, either a row or column is eliminated, and they are marked. The column indices of the front are marked in a way that still lets us perform binary searches.

Each front has a list of descendant fronts that can contribute to the current front. Basically, the prior fronts that are not eliminated are kept in a heap list. In the phase of pivotal column assembly, ParU checks all the heap lists of children of the current front. if lac is less than the larger pivotal index of the current front, then that prior front is going to be (fully or partially) summed in the current pivotal columns. These prior fronts are removed from the heap and added to another data structure. Most of the time, these prior fronts are fully assembled in the current front either into pivotal columns or the rest of the front. Note that the gray part in Figure 2.8 is a single memory

allocation and has a different physical space than the two other parts of the current front. The rest of the prior fronts that do not contribute to the pivotal columns are kept as heaps, and after the assembly, a single heap is made out of all the heaps and is passed to the parent.

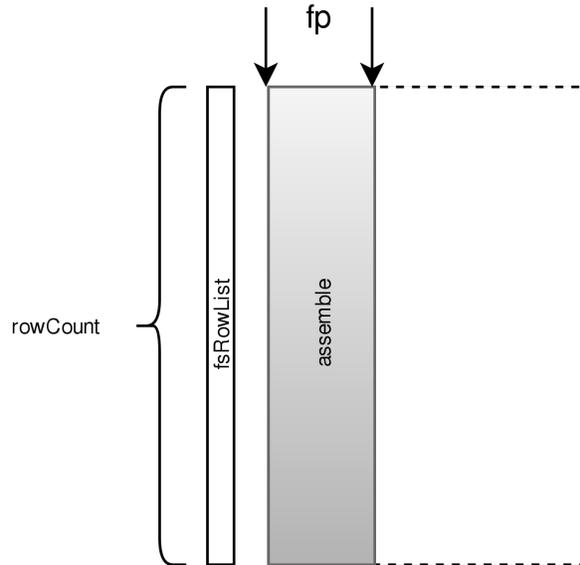


Figure 2.8: Assembly of pivot columns

During the pivotal column assembly, due to the staircase structure of the input, sometimes rows of entire zeros are added to the pivotal columns. Those rows are usually padding zeros of prior contribution blocks. Although there are non-zeros on that rows, they might not appear in the current front. These rows can accumulate in several iterations and add useless work until they are filled. In ParU, these rows are not added to the current front, while they add extra unnecessary flops to the computation. Their prior contribution should remain in a heap, though. Therefore ParU deals differently with this kind of pivotal column.

2.3.3 Pivotal column factorization

When the numerical values of the pivot columns are assembled, we can factorize the pivotal columns, which is called partial factorization. Note that the factorized part (blue part in Figure

2.9) will be a part of the final result. We use the numerical values to update the U part (triangular solve) and compute the Schur complement(matrix multiplication). After computing the Schur complement, both factorized part and the U part will no longer be necessary for the rest of the factorization phase. While ParU uses different memory allocations for each one, and there is no more access to this data in this phase, operating system virtual memory management probably stores it on the disk.

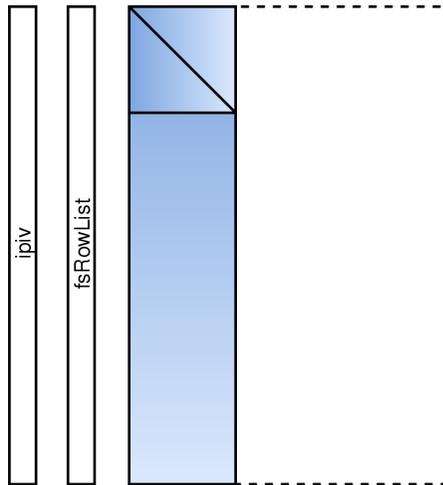


Figure 2.9: Factorize the pivot columns

In ParU, during factorizing the pivotal columns, no other assembly happens. Pivoting can change the size of the current front. However, ParU does not form the rest of the front only until all the factorization of the pivotal columns is completed. A single front can be very large, and we might need several stages to factorize it. UMFPACK grows the front one column at a time and intends to keep the front as small as possible while forming the whole front. In ParU, on the other hand, we grow the factorizing of the pivotal column by a set of pivot columns or panels. The user can change the number of columns of a panel using the control option `panel_width`.

The contribution block's number of columns is unknown until we pick all the pivots. When ParU factorizes a panel, it knows the rows of the panel that is a subset of rows of that front. It

can also compute the number of columns related to that panel. ParU has an upper bound for that (QR upper bound), and it can reallocate the memory if it needs to grow the size of the contribution block. Figure 2.10 shows an active front with three panels, one of which is already factorized. The current front's size is unknown until all the pivots are chosen. The pivots are chosen not only based on values but also based on sparsity. UMFPACK, after choosing a pivot, grows the front as needed and updates the sparsity degree of rows. In ParU, we do not permute columns, so there is no need to keep the degree of columns. Moreover, updating the degree after choosing a single pivot might not be efficient for a parallel implementation.

The parallel algorithm of choosing pivots and updating the degrees is thus a challenge. ParU updates degrees and the sizes of the front after factorizing each panel. On the other hand, while ParU fixes the columns first, it has the exact number of rows of the active front. Compared to UMFPACK, ParU's work is simpler in this part, while it only needs to grow the front just by one dimension. In such an implementation, we can still exploit zeros (Figure 2.10) on the staircase structure, no matter how we allocate memory for the contribution block. Note that although ParU computes the front size; it does not form it explicitly. It only updates the degrees of rows and updates the list of columns contributing to this front at this stage.

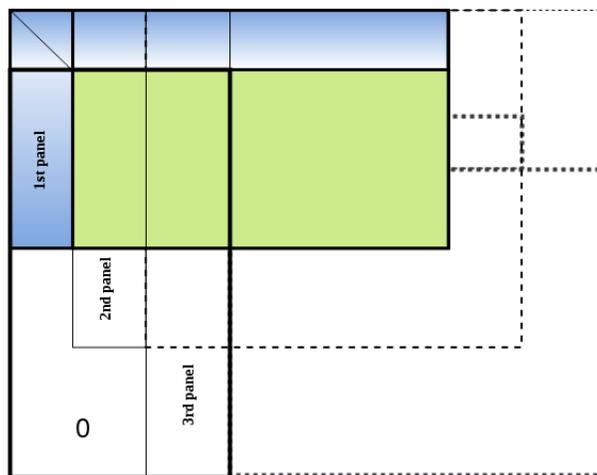


Figure 2.10: Panel factorization

2.3.4 U-part assembly and update

For the active front, we can have the mathematical expression in equation 2.1:

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12}^T \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12}^T \\ a_{21} & A_{22} \end{bmatrix} \quad (2.1)$$

By factorizing the pivotal column of the active front, two equations are solved (l_{11} and l_{21} are blue part of Figure 2.9): $l_{11} \times u_{11} = a_{11}$ and $l_{21} \times u_{11} = a_{21}$.

After the factorization of pivotal columns, ParU has to assemble the numerical values of u_{12} (Figure 2.11), then it has to solve a triangular equation: $l_{11} \times u_{12} = a_{12} \rightarrow u_{12} = a_{12}/l_{11}$.

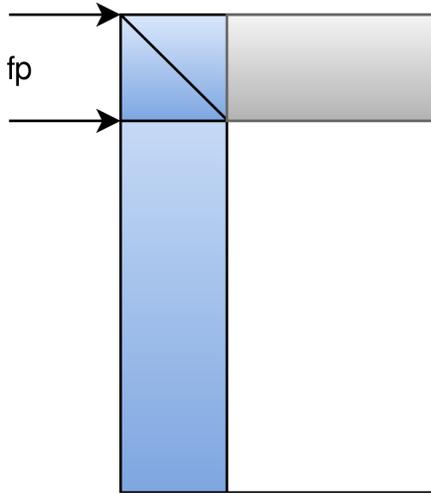


Figure 2.11: Assembling U part numerical values

The only remaining part of the equation is solving $l_{21} \times u_{12} + L_{22} \times U_{22} = A_{22} \rightarrow L_{22} \times U_{22} = A_{22} - l_{21} \times u_{12}$.

This would be part of the contribution block that will be used in computation for future fronts. The blue parts in Figure 2.12 are computed, and we will need the green part for future (future prior fronts). Triangular solve and matrix-matrix multiplication here is computationally intensive, and

they are a good candidate to be parallelized. In fact, in most cases, this matrix-matrix multiplication is the most time-consuming part of factorization. The challenge in ParU arises from the fact that you can not necessarily use all the computational cores for the current front. There might be other fronts that also have similar computations. In other words, we have nested parallelism in ParU, and we will discuss how we address the problem in section 2.5.1.

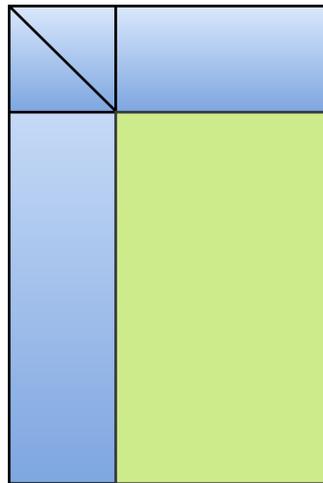


Figure 2.12: Applying $-l_{21} \times u_{12}$

2.3.5 Contribution block assembly

Prior front assembly on the contribution block can have a very nice effect on the performance. However, it is not mandatory to get the correct result. The only mandatory part is assembling the pivotal columns. Especially in an unsymmetric algorithm, we do not only have a full prior front assembly; we can have partial column/row assembly as well. The algorithm should avoid spending too much time searching for intersections of prior fronts. In UMFPACK, row/column tuple lists (list of ordered pairs) are used for finding the intersection between the current front and prior fronts. In ParU, while we no longer have the column tuples, other algorithms are designed and implemented.

UMFPACK does four passes on the columns and the rows of the current front to know how to

deal with prior contribution blocks. In ParU, we have row tuples, but for the columns, we have two other data structures: a list of elements that contributed to the pivotal columns and a list of heaps of live children of the current front.

Five possible conditions can happen for the current front relative to a prior front. One possible condition is where a prior front has no column or rows in common with the current front, so there can be no assembly. Other cases are depicted in Figures 2.13, 2.14, 2.15, and 2.16.

The most important condition is that all rows and columns of the prior front are in the current front (Figure 2.13). Therefore, we can assemble the entire prior front into the current front. The memory space of the prior front can be released. One instance of this situation is when a prior front contributes to at least one pivotal column and one pivotal row. In ParU, we have the list of the elements that contribute to the pivotal column; if one of them contributes to the pivotal rows, it would be the case. In practice, most of the instances of these cases are like it, and it is simple to find. This case can also happen if the prior front shares all the rows and columns with the current front. However, detecting it requires searching in the columns.

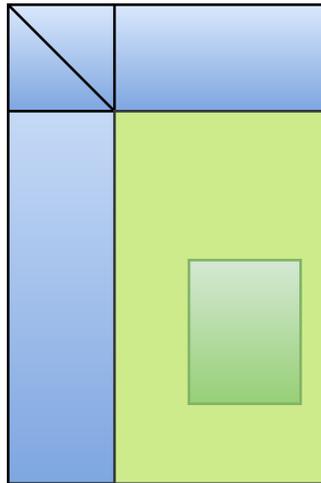


Figure 2.13: The prior contribution block is within the current front

It is also possible that only a set of rows (Figure 2.14) or a set of columns (Figure 2.15) of

a prior front is the subset of the current front, and there are other rows or columns that cannot be assembled to the current front. Elements that contribute to the pivotal columns share all the columns to the current front (Figures 2.14 and 2.13)

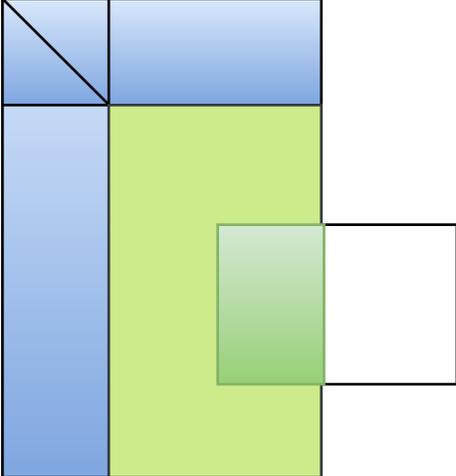


Figure 2.14: Prior contribution block rows are a subset of current front rows

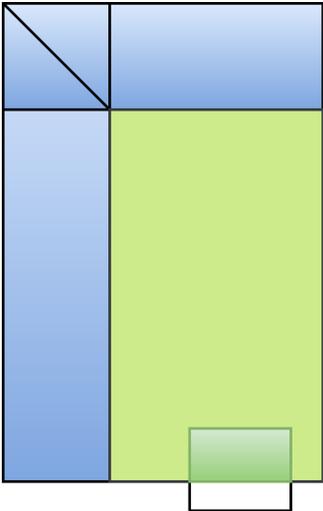


Figure 2.15: Prior contribution block columns are a subset of current front columns

Finally, it is also possible that both rows and columns of the prior contribution block has some rows and columns in common with the current front, but not all of them (Figure 2.16). In this case, a part of the prior contribution block can be assembled, however, since it is hard to manage such an assembly, we prefer not to (like UMFPACK).

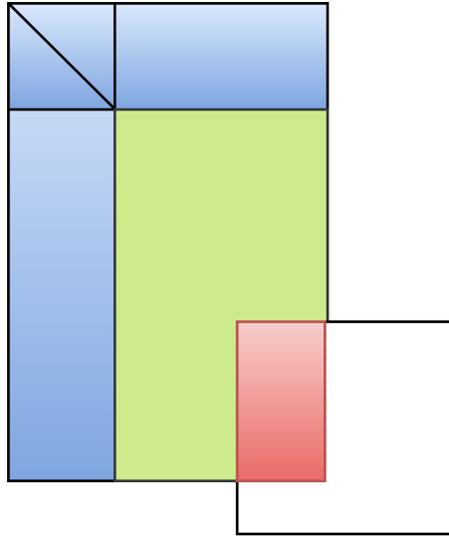


Figure 2.16: Prior contribution block rows and columns have rows and columns in common with the current front (No assembly)

2.3.5.1 *ParU* prior front assembly

In *ParU*, other than the list of the elements contributing to the pivotal columns, we flag the elements that contribute to the pivotal rows. In the prior front assembly phase of the current front, first, all the elements that contributed to the pivotal columns are tested. If they also contributed to the pivotal rows, they can be fully assembled in the current front. Otherwise, they may contribute to several columns. After probing elements that contribute to the pivotal columns, *ParU* checks some of the prior fronts that are in a heap. The algorithm that explains how those heaps are finalized is shown in Algorithm 2.3.5.2.

As we mentioned earlier, the list of columns of the fronts is sorted, and we can use binary search to find a specific column of the current front in the prior front in $\mathcal{O}(\log n)$. To be able to

look up the columns faster, ParU uses a simple hash that takes an expected time of $\mathcal{O}(1)$. We make sure that searching for columns in ParU does not take more than $\mathcal{O}(\log n)$.

For the cases that partially contribute to the columns, we use an incremental algorithm in ParU. The pseudo-code of this algorithm is shown in Algorithm 2. First, the least active column, lac , of the prior front is tested if it is inside the current front; if lac is inside, this column is assembled and goes to the next column and increases lac of the prior front. The process continues until the first column is detected, which is not in the current front. After that, to limit the searching time, we use a mechanism to punish if the column is not inside the current front or reward if it is.

To do so we define an integer number, $toll$ which is initialized by a constant, line 6 in Algorithm 2. We decrement the toll for each column that is not inside the current front, and for any column within the current front, we increase the toll. This process continues until we visit all the columns or the toll is zero. With this algorithm, if the prior front fully fits the current front, we can fully assemble it, but we also avoid searching too much to find the intersection. A similar algorithm is implemented for the cases that partially contribute only to the rows. ParU, store dense matrices column-oriented and column-by-column assembly is more cache-friendly. Therefore, ParU assembles by columns even for the row assembly.

2.3.5.2 *Finalizing the heap*

A list of live descendent fronts in a heap data structure is passed into the current front, a list of fronts that contributed to the pivotal columns is extracted from them, and the rest of the heap is left intact. During the prior front assembly phase, the heaps are merged together to make a new heap to pass to the parent. ParU does not touch the prior heap if it is not necessary. Heapifying some of the lists can be really expensive. We also decided not to check all the prior elements not to fall into the trap of doing too much computation without any gain.

One of the reasons that ParU waits until the assembly time is that usually, many of the prior elements are already fully assembled, making heapifying cheaper. ParU keeps track of the size of the biggest heap among the children. If the biggest child is empty, there is nothing to be done with the heap.

Algorithm 2: Assemble Columns

Data: Current element, e_c and a prior element e_p which contributes to the columns
//Toll-free zone
//using hash or binary search to find the column

```
1 while column  $lac(e_p)$  is inside  $e_c$  do
2   | assemble the column  $lac(e_p)$  ;
3   |  $lac(e_p) \leftarrow$  next active column
4 end
6 toll  $\leftarrow$  8;
7  $n_c \leftarrow lac(e_p) + 1$ ;
  //Toll zone
8 while toll > 0 do
9   | if column  $n_c$  is inside  $e_c$  then
10  |   | assemble the column  $n_c$ ;
11  |   | toll  $\leftarrow$  toll+1;
12  |   | else
13  |   |   | toll  $\leftarrow$  toll-1;
14  |   | end
15 end
```

If the biggest child is much larger than the rest of the children, it is cheaper not to even look at the elements of the biggest child and add the rest to the biggest child. Adding one element to a heap is $\mathcal{O}(\log n)$. Therefore, the complexity of making a heap this way is $\mathcal{O}(k \log n)$ in which k is the size of the rest of the elements. In this case, the elements of the biggest child are not checked for assembly.

Otherwise, if the size of the biggest child is about the order of the rest of the children, ParU uses the $\mathcal{O}(n)$ algorithm to heapify all the children, and all of the children are checked for the assembly. The list of pivotal elements is already checked for assembly, and it is already not a heap data structure. The remaining pivotal elements are added to the final heap structure one by one using the $\mathcal{O}(\log n)$ algorithm. The algorithm is shown in algorithm 3.

2.3.6 Summary of Numerical Factorization algorithm

The summary of the factorization algorithm is shown in algorithm 4. This algorithm can run from the first front to the last. Any topological order can work and we will explain how it can be

Algorithm 3: Finalizing the heaps

```
Data: Find the biggest heap biggest_Child
if  $\log_2(\text{Size}_{\text{biggest\_Child}}) > (\text{Size}_{\text{biggest\_Child}}/(\text{Size}_{\text{rest}} + 1)) + 1$  then
    //Using  $O(k \log n)$  algorithm
    for  $e_i$  in Rest do
        | Check  $e_i$  for row/col assembly ;
        | Add  $e_i$  to the biggest_Child;
    end
else
    //Using  $O(n)$  algorithm
    for  $e_i$  in All do
        | Check  $e_i$  for row/col assembly ;
    end
    Heapify(All);
end
```

implemented in parallel in Section 1.4.3.

2.4 Solve

Solve phase consists of a forward elimination and a backward substitution. Typically the solve phase is much faster than the factorization. Therefore, in ParU, we decided not to implement front-level parallelism during the solve phase. However, it is possible to implement both in the same manner that we implemented factorization. Forward elimination traverses from the bottom to the top of the tree, and backward substitution traverses the tree from the bottom to the top.

In ParU, we use parallel BLAS for the solve phase and compute from the front 0 to the last front (or vice versa). The data structure of the factors plays an important role in the efficiency of the algorithm of the solve phase. Let's assume that we have n_f number of fronts for solving. ParU partition the input matrix into n_f dense matrices that contain L and U and $n_f - 1$ dense matrices that only contain U . Note that singletons also must be taken into account. The partition of a matrix with three fronts is shown in the Figure 2.17. This figure shows the matrix with L and U as LU . Note that the size of the dense matrices is much smaller than the dimension of the input matrix, but the figure shows the placement of the elements inside the matrix. The number of rows in LU

Algorithm 4: Summary of Numerical Factorization algorithm

Data: Initialize the row elements

for $1 \rightarrow n_f$ **do do**

 //Working on front f_i

$num_{rows} \leftarrow$ Find the set of rows for f_i (See Algorithm 1) ;

 Allocate memory with the size $num_{pivotal_col} \times num_{rows}$;

 Assemble pivotal columns. (See Section 2.3.2) ;

 Factorize the pivotal columns (partial factorization). (See Section 2.3.3) ;

 Allocate memory with the size $num_{pivotal_ccol} \times num_{nonpivotal_col}$;

 Assemble the U-part. (See Section 2.3.4) ;

 Allocate memory with the size $num_{rows} - num_{pivotal_col} \times num_{nonpivotal_col}$;

 Schur complement of U-part and pivotal column part (See Figure 2.12) ;

 Assemble the contribution block (See Section 2.3.5.1) ;

 Finalize the heap (See section 2.3.5.2) ;

end

matrices and the number of columns in U matrices in the figure are smaller than what you see in the figure.

We should consider the matrix's scaling, permutation, and singletons during the solve phase. ParU, like UMFPACK (see 1.4.2, factorize $PRAQ$ into LU and for the solve we have to compute $x = Q * (U \setminus (L \setminus (P * R * b)))$ in MATLAB notation. ParU applies $P * R * b$ first, then computes the forward elimination, then computes the backward substitution, and finally applies Q . After dealing with singletons, forward elimination performs a triangular solve DTRSV for each of the fronts and a DGEMV for those that need it. ParU also has a multiple right-hand side solver that uses DTRSM and DGEMM accordingly. Backward substitution is similar but starts from the last front and goes back to the first front.

2.5 Parallelism

We decided to use OpenMP to parallelize ParU. However, the algorithm is suitable for any kind of shared-memory parallelism as it is. As mentioned, the elimination and task trees are generated during the symbolic analysis phase. For example, in the Figure 2.18, the elimination tree of the matrix `494_bus` from SuiteSparse Matrix Collection [31] is depicted. This tree is generated after

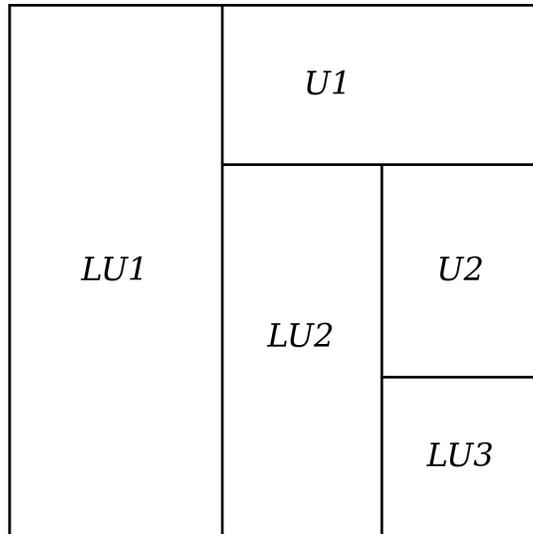


Figure 2.17: ParU factors relative to the input matrix

the amalgamation and staircase structure.

Before looking deeper into the current algorithm implemented in ParU, let us explain the process that led us to the current algorithm. The first parallel implementation used the original elimination tree and was recursively implemented from the root(s). In this implementation, each child is a task that runs in parallel with the siblings, and when all the children are done, the parent can start.

This head recursion implementation is really easy to implement, and it works. However, it has several drawbacks. First, there are always the middle fronts and root(s) waiting for their term of execution, which can be an overhead for the OpenMP tasking queue. This is not an issue for many matrices, but when the matrix has a huge and unbalanced etree, that can cause problems. Second and most importantly, the OpenMP tasking tends to perform the task in the tree level by level. Unbalanced trees will lead to performing the critical path at last, and it significantly reduces the performance. Using `OpenMP priority` does not change this effect much.

The other approach is traversing the tree from the bottom to the top. This approach does not have a long waiting time for some tasks in the queue. The longest queue is at the algorithm's beginning, and the queue's size is the tree's number of leaves. However, some of the leaf tasks

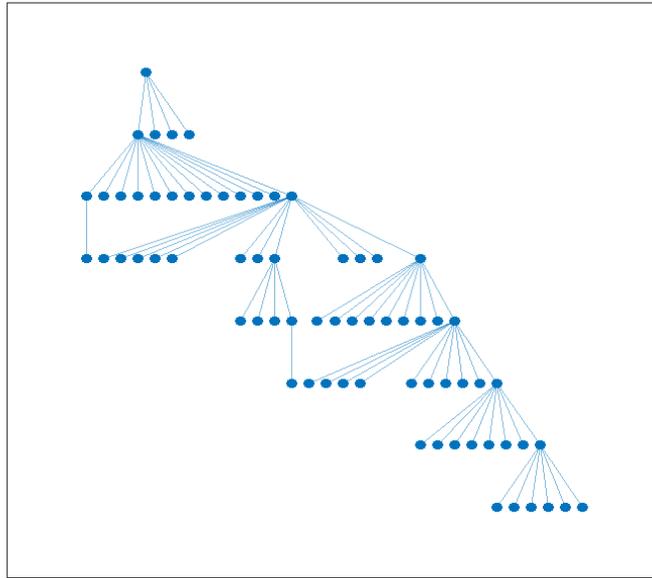


Figure 2.18: Elimination tree of the input matrix

are small, and making a single task for them can reduce performance. Therefore, ParU makes another tree from the etree, which is only made for parallelism purposes. This tree is coarser, and leaves with less work are merged together. The task tree of the etree in Figure 2.18 is shown in Figure 2.19. The chains of the original etree are also merged into a single task. A chain in an etree is a sequence of fronts with only one child. The resulting task tree might still have chains, but ParU does not eliminate secondary chains.

Each node in the task tree can be a front or a group of fronts, and the factorization can be done from the bottom to the top of the tree. The only limitation is that you can not start a task unless you have finished all the children. ParU makes a queue of all the tasks that can be immediately done. ParU, sort these tasks based on their depth in the task tree and start to schedule them using OpenMP tasking. In Figure 2.20, all the tasks that are initially inserted into the queue are shown with a different color.

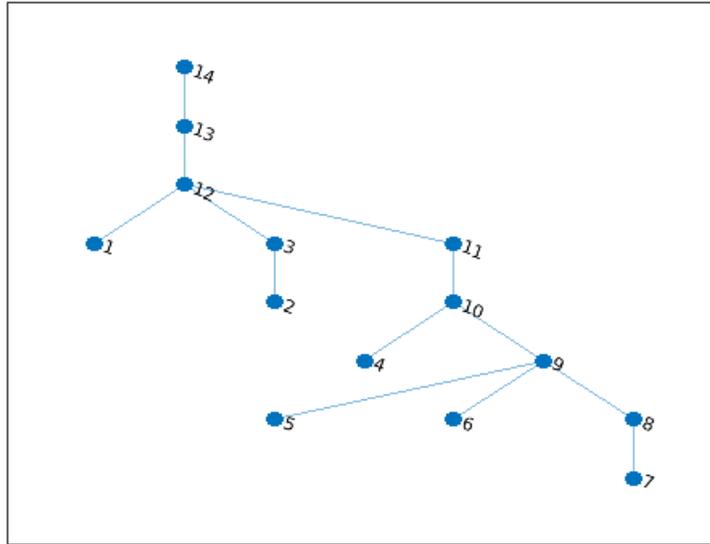


Figure 2.19: Task tree of matrix from Figure 2.18

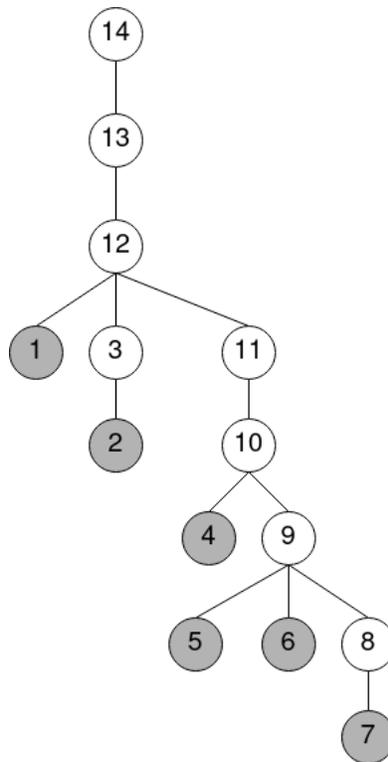


Figure 2.20: Ready fronts to be executed

Once each task finishes its work, the number of remaining children of its parent is atomically decremented by one. If it is the last child, it can execute the parent in a recursive manner. Figure 2.21 uses a darker highlight for the tasks that are already finished. For example, when task 7 is finished, it can put task 8 into the queue. When task 6 is finished, it should wait for siblings 5 and 8 to be able to start task 9. This algorithm is shown in Algorithm 5. The function `Execute` in this algorithm is defined recursively. The recursion is tail recursion, which basically partitions the task tree into smaller trees on the runtime. It is a very flexible way to partition the tree. However, it might partition the tree in different ways each time.

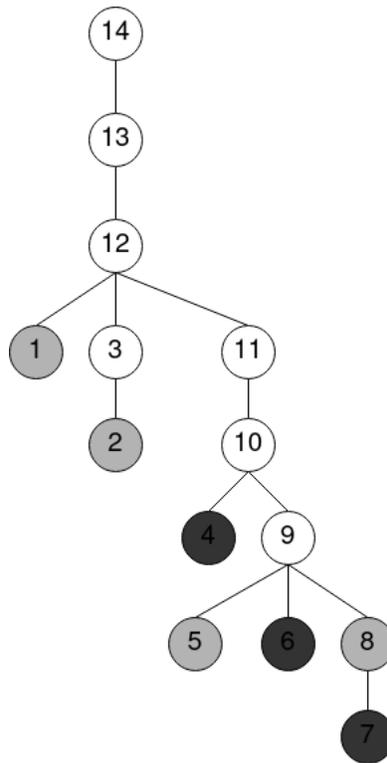


Figure 2.21: The tree in the middle of the execution

An example of how running tasks can partition the tree is shown in 2.22. This partition can change in another run. For example, if task 5 finishes the last between its siblings, 6 and 8, then that is the task that will call their parent 9. There are some details in implementing how the task

Algorithm 5: Parallel task execution

Data: make an array of #task_children (NoC) the tasks with > 1 child

Data: make a $Task_Q$ of tasks with no children

Parallel Region

```
    while enough tasks do
        //look at algorithm 6 for defining enough
         $T_0 \leftarrow Enqueue(Task\_Q)$ ;
        Execute( $T_0$ );
         $P \leftarrow parent(T_0)$ ;
        Atomic
        |    $NoC(P) \leftarrow NoC(P) - 1$ ;
        |    $Num\_rem \leftarrow NoC(P)$ ;
        end
        if  $Num\_rem = 0$  then
            Execute( $parent(T_0)$ );
            //T0 is the last child
        end
    end
end
```

must run. For example, in Figure 2.22 top tasks 1, 12, 13, 14 are the only task remaining, and there is no parallelism among tasks. For some libraries, it is better to jump out of the OpenMP parallel region and execute the last chain with a fully parallel BLAS. The main reason is that many BLAS libraries use OpenMP inside, and controlling the nested parallelism and thread affinity of OpenMP is difficult between the two libraries. This is also shown on Algorithm 6 line 10.

2.5.1 scheduling

Each task contains a series of dense matrix-matrix multiplications and triangular solves, and these tasks can be executed in parallel. Therefore, there is nested parallelism in ParU. Managing nested parallelism is sometimes difficult and can reduce some inner libraries' performance. ParU jumps out of the parallel region when no more parallel task is available. We found that it is more efficient to do this in practice.

BLAS computation takes most of the computation time during factorization, and using the right libraries greatly impacts overall time. In Table 2.1, the analysis of Intel Vtune is shown as a matrix

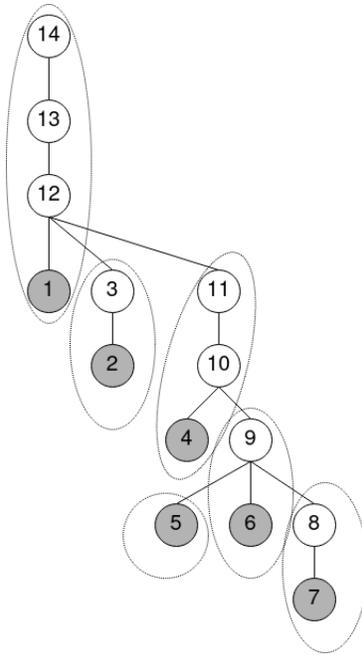


Figure 2.22: An example of how the task can partition the tree

Function	CPU Time: Total
ParU_Factorize	42.3%
cblas_dgemm	29.9%
paru_assemble_all	7.3%
ParU_Analyze	1.1%

Table 2.1: time consuming functions in ParU

being factorized using ParU. Note that functions might overlap. Full analysis of Vtune is shown in Appendix A Figure A.1. Typically the analysis of Vtune shows that most of the time is spent on DGEMM. Computing resource distribution can have a significant effect on the efficiency of the overall program. In the Algorithm 6 the way ParU distributes the tasks for BLAS is shown. The full Vtune analysis is shown in Appendix Figure A.1.

There are several places in the ParU where BLAS is needed. During the pivotal column factorization. DGER is needed, and after each panel DTRSM and DGEMM is needed for the rest of the pivots. A DTRSM and a DGEMM (see section 2.3.4) is needed for the contribution block, and they

Algorithm 6: Scheduling Algorithm for BLAS

Data: make a $Task_Q$ of tasks with 0 child

```
1 if enough parallelism then
2   | while  $tasks > computing\ cores$  do
3   |   Use single thread BLAS
4   | end
5   | while  $tasks > 1$  do
6   |   Use parallel BLAS (some threads)
7   | end
8   | Break out of parallel region;
9   | //Only one chain remained
10  | Use parallel BLAS (all threads) for the last task;
11 else
12 | execute fronts  $0 \rightarrow n_f$  with parallel BLAS
13 end
```

can be large computations. Typically, most floating operations are spent on the DGEMM. DGEMM computes the result of $C = \alpha A \times B + \beta C$, where C is m by n , A is m by k and B is k by n . The sizes of m , n , and k are usually small at the bottom of the tree, and they get larger as we traverse the tree. In ParU, They can get really big, especially for the Schur complement In comparison, UMFPACK does not let the k for BLAS be bigger than a threshold. Therefore, usually, ParU has fewer BLAS operations but bigger ones that are more suitable for contemporary architectures and accelerators.

2.5.1.1 BLAS scheduling

There are two stages in the scheduling that are fairly straightforward. First when there are more active fronts than the total number of cores and second when there is just a single active front. For the prior, we can use a single thread BLAS, and for the latter, we use all the threads for the BLAS operation (Algorithm 6 lines 3 and 11). In practice, even if enough tasks are ready, using parallel BLAS with fewer threads for each task can be beneficial. Users have control over a maximum number of front-level threads. The default number in ParU is less than the maximum number of threads. Therefore the algorithm usually starts with some local BLAS parallelism.

The middle stage can be complicated and greatly impact performance. There are also cases where the computation in the etree is unbalanced, and you start with many active fronts, and only one of them remains after some time. Having an idea of how many fronts are active and how many tasks remain can be difficult. In ParU, we use two global variables `naft` (number of active fronts) and `resq` (number of remaining tasks in the queue) to control it dynamically. These variables are accessed atomically and can help run any BLAS computation with the number of threads needed line 7 Algorithm 6.

The second challenge in scheduling is that you cannot locally control the number of threads in some of the BLAS libraries. For example, if you run two instances of parallel OpenBLAS simultaneously, you get a terrible slowdown. Using Math Kernel Library (MKL), you can set the number of threads locally. However, setting thread affinity while using MKL can greatly impact performance.

If ParU is using MKL and in the area of the tree that is using parallel BLAS but with not all the threads, the number of threads for that specific BLAS is the total number of threads divided by `naft`. If the BLAS library is anything besides MKL, ParU uses OpenMP tasking and makes enough tasks with a single-threaded BLAS. This strategy might not be the best solution, while the size of the BLAS is also important in scheduling threads for them. However, it is difficult to have a global scheduler that knows everything about all the running BLAS and schedules them optimally. Moreover, that scheduler can be a bottleneck of the algorithm.

2.5.2 Other Parallelism Options

There are many parallelism options in the ParU source code. However, there are not many cases that show a good performance improvement. The most important parallelism in ParU is among fronts and BLAS parallelism. ParU also uses parallelism for full assembly of a prior front. The partial front assembly is tested for parallelism, but the cost of making threads is usually more than the gain of parallelism.

There is not much parallelism to exploit during symbolic analysis. The algorithms for parallelism during the symbolic analysis are often complicated and mostly do not show a good perfor-

mance. Moreover, most of the work during symbolic analysis is done inside UMFPACK. If the input array size is large enough, the only case that shows a good result is the permutations.

While all the rows are independent, that part is fully parallel in the initializing rows and fronts. During the factorization, if there is enough active front, the dominating parallelism is among fronts. When there are more computing cores than active fronts, ParU has parallel BLAS and parallel assembly. During panel factorization, the only profitable way of parallelism is using vectorization, which is always used regardless of the number of active fronts.

The only parallelism ParU uses during the solve phase is the parallelism within the BLAS libraries. As we mentioned, it is possible to have front-level parallelism, but it is left as future work. There are some parallelism options in this phase, but in practice, it is better only to use parallelism inside the BLAS library.

2.6 Performance comparison with UMFPACK

ParU and UMFPACK have different algorithms for their numerical factorization, which can greatly impact their performance. One difference is that UMFPACK changes the column ordering during numerical factorization, while ParU does not change the column permutation. The other important difference is that ParU does not update the row degrees after each pivot. Therefore the minimum degree ParU finds for pivots other than the first pivot in the panel can be stale. UMFPACK uses an approximate minimum degree for both rows and columns. Our results show that the quality of order does not suffer much when you update row degrees less frequently, which is preferable for parallelism.

To compare the difference in parallelism, let us assume that ParU and UMFPACK use the same elimination tree and the same permutation, which is not usually realistic but simplifies this comparison. Let us assume we have a balanced elimination tree with a root, two children, and two computational cores. We also assume that each front takes 10 seconds to finish the computation. Therefore, it would take 30 seconds to finish everything in a single-core machine.

In multifrontal sparse solvers, most of the time is usually spent on matrix-matrix multiplication. ParU supports only double-precision now, so the routine it calls is DGEMM. DGEMM is one of

the routines that are massively parallel and can be effectively parallelized. UMFPACK executes one single front at a time and calls parallel DGEMM. ParU, on the other hand, calls single-threaded DGEMM while there is enough work for all threads, and it starts to call parallel DGEMM when there are more computational cores available. If the parallel portion of each front is 80% of the computation, UMFPACK can finish each front in 6 seconds. Therefore the total time for UMFPACK would be 18 seconds. This is an optimistic assumption for UMFPACK since we are ignoring the case when the elimination tree has many tiny frontal matrices.

ParU, on the other hand, can execute the children in parallel (10 seconds) and then use both cores to finish the root in 6 seconds. Therefore, in theory, ParU needs 16 seconds to finish factorization. This example shows the theoretical limits of performance of ParU versus UMFPACK, and it is basically a generalization of Amdahl's law in an elimination tree.

In practice, ParU and UMFPACK have different permutations that can lead to different computations. Moreover, the shape of the elimination tree is not usually balanced, which makes the scheduling for ParU difficult. There are also cases that have small fronts, and the parallel portion in each front is less, and ParU has a better strategy for parallelism. Consequently, it is hard to predict the performance of ParU versus UMFPACK. Typically for unsymmetric matrices, the tree is more balanced, and ParU shows a nice performance.

To perform well, ParU typically favors bushy trees. Therefore, selecting the appropriate preordering can influence how well the algorithm performs. Preordering is carried out through the UMFPACK interface. While using METIS [19] frequently results in a more suited tree, the symbolic analysis process may take longer. It may be a good idea to call METIS for symbolic analysis if the user wants to perform the symbolic analysis only once and performs many factorizations. The other two are COLAMD [32] for unsymmetric cases and AMD [33] for symmetric cases. Additionally, the user can send several right-hand sides as a single dense matrix to ParU or call symbolic analysis and factorization once and solve multiple times.

Even though ParU is an unsymmetric algorithm, it adopted UMFPACK's *symmetric mode*. When the pattern of the matrix is close to symmetric, then UMFPACK uses the symmetric mode.

ParU and UMFPACK do not have a specific design for symmetric matrices, but they strongly prefer diagonal entries in symmetric cases. If the diagonal entry has a small relative magnitude, the algorithm can still choose a pivot other than the diagonal element. However, it strongly favors the diagonal entry since it performs better for symmetric matrices. However, given the overhead of the unsymmetric matrix data format, it is still not as effective as symmetric methods like MUMPS [22]. UMFPACK symbolic analysis finds the symmetry of the matrix, 1 for a symmetric pattern matrix and 0 for an unsymmetric matrix. Our experiments show that even with symmetry of about 0.3 in the pattern, it is better to use the symmetric mode. Note that both UMFPACK and ParU cannot use numerical symmetry.

3. PARU CODE STRUCTURE AND RESULTS

ParU is a parallel sparse direct solver. This package uses OpenMP tasking for parallelism. ParU calls UMFPACK for the symbolic analysis phase, after that, ParU itself does some symbolic analysis, and then the numeric phase starts. The numeric computation is a parallel task phase using OpenMP, and each task calls parallel BLAS, i.e. nested parallelism. After that, the solve phase can be called with either a single right-hand side or multiple right-hand sides. The performance of BLAS has a heavy impact on the performance of ParU. However, depending on the input problem, the parallel performance of the BLAS can sometimes have a smaller effect on the performance of ParU.

The details on how to install ParU is on `ParU/Doc/paru_usser_guide.pdf`.

3.1 C/C++ Example

The C++ interface is written using only real matrices. The simplest function computes the MATLAB equivalent of $x=A\backslash b$ and is almost as simple: Below is a simple C++ program that illustrates the use of ParU. The program reads in a problem from `stdin` in MatrixMarket format [34], solves it, and prints the norm of A and the residual. Some error testing code is omitted to simplify showing how the program works. The full program can be found in `ParU/Demo/paru_demo.cpp`.

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

    //~~~~~Reading the input matrix and test if the format is OK~~~
    // start CHOLMOD
    cc = &Common;
    int mtype;
```

```

cholmod_l_start(cc);

// A = mread (stdin) ; read in the sparse matrix A
A = (cholmod_sparse *)cholmod_l_read_matrix(stdin, 1, &mtype, cc);
//~~~~~Starting computation~~~~~
ParU_Control Control;
ParU_Ret info;
info = ParU_Analyze(A, &Sym, &Control);
ParU_Numeric *Num;
info = ParU_Factorize(A, Sym, &Num, &Control);
double my_time = omp_get_wtime() - my_start_time;
//~~~~~Test the results ~~~~~
Int m = Sym->m;
if (info == PARU_SUCCESS)
{
    double *b = (double *)malloc(m * sizeof(double));
    double *xx = (double *)malloc(m * sizeof(double));
    for (Int i = 0; i < m; ++i) b[i] = i + 1;
    info = ParU_Solve(Sym, Num, b, xx, &Control);
    printf("Solve time is %lf seconds.\n", my_solve_time);
    double resid, anorm;
    info = ParU_Residual(A, xx, b, m, resid, anorm, &Control);
    printf("Residual is |%.2lf| and anorm is %.2e and rcond is %.2e.\n",
        resid == 0 ? 0 : log10(resid), anorm, Num->rcond);
    free(b);
    free(xx);
}
//~~~~~End computation~~~~~
Int max_threads = omp_get_max_threads();
BLAS_set_num_threads(max_threads);

//~~~~~Free Everything~~~~~
ParU_Freenum(&Num, &Control);

```

```

    ParU_Freesym(&Sym, &Control);

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

```

3.1.1 C/C++ Syntax

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

1. `ParU_Version`: return the version of the ParU package you are using.
2. `ParU_Analyze`: Symbolic analysis is done in this routine. UMFPACK is called here and after that, some more specialized symbolic computation is done for ParU. `ParU_Analyze` is called once and can be used for different `ParU_Factorize` calls for the matrices with the same pattern.
3. `ParU_Factorize`: Numeric factorization is done in this routine. Scaling and making Sx (scaled and staircase structure) matrix, computing factors, and permutations are here. `ParU_Symbolic` structure which is computed in `ParU_Analyze`, is input in this routine.
4. `ParU_Solve`: Using symbolic analysis and factorization phase output to solve $Ax = b$. In all the solve routines `Num` structure must come with the same `Sym` struct that comes from `ParU_Factorize`. This routine is overloaded and can solve different systems. It has versions that keep a copy of `x` or overwrite it. Also, it can solve multiple right-hand side problems.
5. `ParU_Freenum`: frees the numerical part of factorization.

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

Table 3.1: ParU user-controllable variables

6. ParU_Freesym: frees the symbolic part of factorization.

3.1.2 Details of the C/C++ Syntax

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

1. SuiteSparse/ParU/Include/ParU.hpp describes each C++ function. Only double and square matrices are supported.
2. SuiteSparse/ParU/Include/ParU.h describes the C-callable functions.

There are C/C++ options to control ParU, which is an input argument to several routines. When you make ParU_Control object, it is initialized with default values. The user can change the values. The list of control options are shown in Table 3.1. Here are the details of the list of controls:

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

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

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

3.2 Requirements and Availability

ParU requires several Collected Algorithms of the ACM: CHOLMOD [35, 36] (version 1.7 or later), AMD [15, 33], COLAMD [37, 38] and UMFPACK [1] for its ordering/analysis phase and for its basic sparse matrix data structure, and the BLAS [39] for dense matrix computations on its frontal matrices. An efficient implementation of the BLAS is strongly recommended, either vendor-provided (such as the Intel MKL, the AMD ACML, or the Sun Performance Library) or other high-performance BLAS such as those of [28]. Note that while ParU uses nested parallelism heavily the right options for the BLAS library must be chosen.

The use of OpenMP tasking is optional, but without it, only parallelism within the BLAS can be exploited (if available).

3.3 Code Anatomy

The user callable functions which is described in section 3.1.1 is in `ParU/Include/ParU.hpp`. Several overloaded functions in that file are usually for the distinction between a single right-hand side and multiple right-hand sides. In ParU, the convention is to use capital `ParU` for the beginning of the name of user callable routines and use `paru` for the rest.

Source files are in the directory of `ParU/Source`. `/Source/paru_analyze` is the file that contains `ParU_Analyze` routine and the symbolic analysis are done in this file. This file is explained in section 2.2. `paru_factorize.cpp` is the file where numerical analysis occurs, either sequential or parallel. In Figure 3.1 the main routine in this file is shown and how it calls other routines to do the factorization. Doxygen makes this Figure, and you can make this using the Doxygen in `ParU/Doc/`.

`paru_init_rowFronts` that is called before `paru_front` does the initializations, scale the matrix, form numerical values for singletons, and S matrix and form initial fronts.

3.4 Experimental Results

ParU is written in C++ and uses OpenMP tasking; it has primarily been tested on Linux platforms. Microsoft Windows only supports OpenMP 2.0, so testing ParU on Windows computers are not available. The package does work on Windows Subsystem Linux (WSL). ParU is appropriate for unsymmetric-pattern matrices because it is built with rectangular fronts. Although it can also solve symmetric systems, a symmetric technique is more effective.

The computer on which we do our testing has two NUMA nodes, a 12-core (24-thread) Intel Xeon E5-2695 processor running at 2.40GHz, 30MB of cache per core, and a total of 792GB of memory.

The SuiteSparse Matrix Collection is where we acquired the matrices we used for the studies [31, 40]. The matrices mentioned in Table 3.2 are selected to be square, asymmetric, and have full sparse rank. These are all the matrices in the SuiteSparse Matrix Collection that are real, and square, have less than 65 percent pattern symmetry, have less than 15 percent numeric symmetry, and have more than one million nonzero entries. We expect ParU to perform well on these matrices, while they do not have much symmetry and are big enough. The MATLAB code to obtain this list of matrices is shown in Listing 1. Table 3.2 shows the name of the matrix, the pattern symmetry, the numerical symmetry, and the number of nonzeros of each matrix.

We compare ParU, UMFPACK[23, 3], SuperLU_MT[10] and MUMPS[22]. The time reported is the total time for symbolic analysis, numerical factorization, and solve with a single right-hand side for all of these solvers utilizing AMD for reordering. Table 3.3 contains the comparison. In this table, the best runtimes are indicated by bold entries.

In our tests, MUMPS performs well, especially for the matrices with an high degree level of symmetry, as shown in Table 3.3. MUMPS always utilizes a symmetric method and is not intended for unsymmetric scenarios. The algorithm is significantly simpler when a symmetric approach is used since the fronts are square, no partial assembly is required. The symmetric mode in ParU

```

index = ssget ;

f = find (index.nrows == index.ncols & ...
index.sprank == index.ncols & ...
index.sprank == index.nrows & ...
index.isReal & ~index.isGraph & ...
index.numerical_symmetry < .15 &...
~index.posdef & ...
index.nnz >=1e6 & ...
index.pattern_symmetry <= .65 ) ;

```

Listing 1: MATLAB script to obtain the list in Table 3.2

derives from UMFPACK. ParU and UMFPACK do not have a specific design for symmetric matrices, but they significantly favor diagonal entries in symmetric cases. The procedure changes little; ParU and UMFPACK strongly prefer the diagonal entry, which works better for symmetric matrices. However, it is not as good as symmetric algorithms while we have the burden of the data structure of unsymmetric matrices.

A group of problems in Table 3.3 shows a good performance with SuperLU. These are optimal power flow problems, and KLU [41] does better addressing them. They are also strongly reducible to block triangular form. Using this form is the best strategy for these matrices. But none of the techniques discussed here can accomplish that. However, SuperLU, which is a left-looking problem, does a better job than multifrontal methods here.

The two biggest problems are in the VLSI group, and they are huge problems. SuperLU did not finish either of the problems after 7 hours. ParU reported that it does not have enough memory for either of these problems using AMD. MUMPS could solve the smaller one and run out of memory for the bigger one. Note that these two problems are fairly symmetric-pattern. ParU, UMFPACK, and MUMPS do better on these problems using METIS.

ParU takes the symbolic analysis phase of UMFPACK and adds a few extra steps to it, which makes ParU's symbolic analysis phase a little slower than UMFPACK's. ParU and UMFPACK's performance can be considerably different because of the different paths in numerical factoriza-

Table 3.2: List of matrices

Name	Pattern symmetry	Numerical symmetry	NNZ
rim	63.90%	0%	1,014,951
TSOPF_RS_b39_c30	5.90%	0%	1,079,986
twotone	24.50%	10.60%	1,224,224
std1_Jac2	0%	0%	1,248,731
mac_econ_fwd500	6.00%	0.60%	1,273,389
std1_Jac3	0%	0%	1,455,848
TSOPF_RS_b300_c1	1.00%	0%	1,474,325
lhr71	0.20%	0%	1,528,092
Zd_Jac2	0%	0%	1,642,833
av41092	0.10%	0%	1,683,902
Zd_Jac6	0%	0%	1,711,983
crashbasis	55.00%	0%	1,750,416
bbmat	53.00%	0.10%	1,771,722
Zd_Jac3	0%	0%	1,916,152
mc2depi	0%	0%	2,100,225
TSOPF_RS_b300_c2	1.00%	0%	2,943,887
TSOPF_RS_b678_c1	0.40%	0%	4,396,289
TSOPF_RS_b300_c3	1.00%	0%	4,413,449
Chebyshev4	30.20%	0%	5,377,761
Hamrle3	0%	0%	5,514,242
pre2	33.20%	6.50%	5,959,282
TSOPF_RS_b2052_c1	0.40%	0%	6,761,100
torso1	42.40%	0%	8,516,500
TSOPF_RS_b678_c2	0.40%	0%	8,781,949
TSOPF_RS_b2383	0.20%	0%	16,171,169
vas_stokes_4M	46.40%	11.40%	131,577,616
stokes	47.30%	12.90%	349,321,980

tion. One difference is that whereas UMFPACK changes the column ordering during numerical factorization, ParU does not alter the column permutation. The fact that ParU does not update the row degrees following every single pivot is another important difference. ParU's minimum degree may consequently be out of date for pivots other than the first pivot on the panel. Our results show that changing row degrees less frequently is actually preferred for parallelism and does not

Table 3.3: Comparison between different methods (total time for analysis, factorization, and solve, in seconds)

Name	SuperLU	MUMPS	UMFPACK	ParU
rim	0.40	0.24	0.74	4.70
TSOPF_RS_b39_c30	0.31	0.35	0.30	0.27
twotone	1.85	1.34	0.93	0.68
std1_Jac2	1.20	1.31	0.66	0.55
mac_econ_fwd500	-	4.21	6.90	4.96
std1_Jac3	1.10	1.49	0.65	0.56
TSOPF_RS_b300_c1	0.39	0.36	0.53	0.42
lhr71	0.56	0.81	0.72	0.48
Zd_Jac2	1.06	1.47	0.79	0.65
av41092	2.91	1.77	3.42	3.25
Zd_Jac6	0.97	1.66	0.76	0.65
crashbasis	1.74	1.43	1.53	1.19
bbmat	3.56	1.08	3.63	6.48
Zd_Jac3	1.06	1.71	0.79	0.70
mc2depi	2.93	3.59	6.56	3.84
TSOPF_RS_b300_c2	0.52	0.71	1.09	0.88
TSOPF_RS_b678_c1	1.19	1.32	3.99	2.01
TSOPF_RS_b300_c3	0.74	1.10	1.67	1.39
Chebyshev4	-	5.84	191.36	5.80
Hamrle3	-	114.86	38.61	24.94
pre2	73.83	9.36	46.95	42.60
TSOPF_RS_b2052_c1	1.96	2.31	6.66	3.35
torso1	2.58	3.48	12.03	4.94
TSOPF_RS_b678_c2	1.42	2.61	5.22	4.17
TSOPF_RS_b2383	4.67	6.66	56.42	66.79
vas_stokes_4M	-	1920.63	-	-
stokes	-	-	-	-

significantly degrade the quality of the result.

In practice, matrix-matrix multiplications take up most of a multifrontal factorization's processing time, and UMFPACK can use parallel BLAS. Therefore, as you can see in Table 3.3, UMFPACK performs similarly to other parallel solvers and occasionally performs better than others. Comparing two solvers is challenging since they each produce different orderings, which

leads to distinct fill-in entries and flops. UMFPACK might be quicker for smaller matrices and outperform ParU according to the etree.

The overhead of parallelism often has the potential to lower performance. The algorithm may get more complicated with nested parallelism. As a result, ParU first determines whether having front-level parallelism is a good idea. If there are no more parallel fronts accessible, the algorithm also exits the nested parallel region. In that case, the efficient use of parallel BLAS benefits the algorithm.

In Table 3.4, you can see a complete comparison between ParU and UMFPACK. In this table, symbolic analysis, numerical factorization, and solve times are depicted in different columns. These results also come from the AMD preordering for both ParU and UMFPACK. As you can see, typically, the symbolic analysis phase of ParU takes longer than UMFPACK. However, numerical factorization and solve time of the ParU can be better. The numerical factorization of ParU is described in detail in this thesis. The solve phase is usually a very small portion of the total time. However, ParU's data structure of the results is more straightforward than UMFPACK, and that can make the solve phase faster.

In Table 3.5, you can see a complete comparison between ParU and UMFPACK using METIS preordering. This table is like Table 3.4. However, we add two larger matrices that run out of memory with AMD preordering. Using METIS, both UMFPACK and ParU can solve `vas_stokes_4M`, and ParU can solve `stokes`.

Table 3.4: ParU and UMFPACK detailed comparison using AMD or COLAMD (run time in seconds)

Name	ParU_sym	ParU_fact	ParU_solve	ParU_tot	UMF_sym	UMF_fact	UMF_solve	UMF_tot
rim	0.10	4.39	0.21	4.70	0.05	3.54	0.13	3.72
TSOPF_RS_b39_c30	0.11	0.15	0.01	0.27	0.09	0.20	0.02	0.30
twotone	0.13	0.51	0.04	0.68	0.10	0.74	0.09	0.93
std1_Jac2	0.39	0.14	0.02	0.55	0.40	0.24	0.03	0.66
mac_econ_fwd500	0.58	4.19	0.19	4.96	0.55	5.86	0.49	6.90
std1_Jac3	0.39	0.16	0.01	0.56	0.40	0.23	0.02	0.65
TSOPF_RS_b300_c1	0.30	0.11	0.01	0.42	0.29	0.20	0.02	0.51
lhr71	0.25	0.22	0.02	0.48	0.23	0.47	0.02	0.72
Zd_Jac2	0.48	0.14	0.02	0.65	0.49	0.27	0.03	0.79
av41092	0.22	2.88	0.14	3.25	0.19	2.99	0.24	3.42
Zd_Jac6	0.46	0.18	0.01	0.65	0.47	0.28	0.02	0.76
crashbasis	0.26	0.85	0.07	1.19	0.22	1.33	0.27	1.81
bbmat	0.27	6.11	0.11	6.49	0.25	4.17	0.43	4.85
Zd_Jac3	0.50	0.17	0.03	0.70	0.50	0.28	0.01	0.79
mc2depi	0.59	2.96	0.29	3.84	0.54	5.61	0.41	6.56
TSOPF_RS_b300_c2	0.65	0.21	0.02	0.88	0.62	0.42	0.05	1.09
TSOPF_RS_b678_c1	1.72	0.28	0.02	2.01	1.71	2.21	0.08	3.99
TSOPF_RS_b300_c3	1.04	0.32	0.02	1.39	0.94	0.63	0.10	1.67
Chebyshev4	1.07	4.52	0.21	5.80	0.97	3.10	0.24	4.31
Hamrie3	1.95	21.86	1.12	24.94	1.86	34.34	2.41	38.61
pre2	3.59	37.19	1.80	42.59	2.85	43.31	1.95	48.11
TSOPF_RS_b2052_c1	2.96	0.37	0.02	3.35	2.87	3.66	0.14	6.66
/Norris/torso1	0.95	3.93	0.06	4.94	0.77	4.75	0.20	5.73
TSOPF_RS_b678_c2	3.56	0.58	0.03	4.17	3.51	1.55	0.15	5.22
TSOPF_RS_b2383	1.93	64.62	0.24	66.79	1.49	53.03	1.90	56.42

Table 3.5: ParU and UMFPACK detailed comparison using METIS (run time in seconds)

Name	Paru_sym	Paru_fact	Paru_solve	Paru_tot	UMF_sym	UMF_fact	UMF_solve	UMF_tot
rim	0.34	0.33	0.02	0.69	0.26	0.42	0.04	0.72
TSOPF_RS_b39_c30	0.45	0.16	0.01	0.62	0.44	0.29	0.02	0.75
twotone	1.32	0.59	0.03	1.94	1.37	0.86	0.07	2.30
std1_Jac2	0.75	0.14	0.02	0.91	0.78	0.30	0.03	1.11
mac_econ_fwd500	4.23	1.30	0.36	5.89	4.44	3.55	0.26	8.25
std1_Jac3	0.79	0.16	0.01	0.96	0.80	0.32	0.02	1.14
TSOPF_RS_b300_c1	0.73	0.17	0.01	0.91	0.71	0.35	0.03	1.10
lhr71	1.75	0.21	0.02	1.98	1.89	0.59	0.02	2.50
Zd_Jac2	0.94	0.18	0.02	1.14	0.94	0.37	0.04	1.35
av41092	5.26	0.80	0.03	6.09	5.25	2.19	0.10	7.54
Zd_Jac6	0.95	0.18	0.02	1.16	0.98	0.40	0.04	1.41
crashbasis	1.79	1.27	0.14	3.20	1.89	1.66	0.15	3.69
bbmat	1.09	2.70	0.17	3.97	1.08	2.25	0.23	3.56
Zd_Jac3	0.98	0.19	0.02	1.18	1.00	0.39	0.03	1.42
mc2depi	5.13	1.62	0.97	7.73	5.31	4.77	0.38	10.46
TSOPF_RS_b300_c2	1.53	0.29	0.02	1.84	1.54	0.76	0.07	2.37
TSOPF_RS_b678_c1	4.18	0.57	0.02	4.77	4.02	2.05	0.12	6.19
TSOPF_RS_b300_c3	2.65	0.42	0.03	3.10	2.60	1.09	0.12	3.81
Chebyshev4	2.45	7.75	0.30	10.50	2.47	4.18	0.40	7.06
Hamrle3	17.33	13.34	1.29	31.96	18.53	21.20	1.70	41.43
pre2	69.45	10.53	1.46	81.44	69.96	18.67	1.20	89.83
TSOPF_RS_b2052_c1	7.34	0.89	0.04	8.26	7.10	3.30	0.15	10.55
torso1	3.65	3.02	0.07	6.75	3.34	4.46	0.25	8.05
TSOPF_RS_b678_c2	8.52	0.94	0.06	9.53	8.38	2.43	0.19	11.00
TSOPF_RS_b2383	24.32	2.52	0.24	27.07	24.76	9.91	0.51	35.18
vas_stokes_4M	202.35	2354.80	23.50	2580.68	214.10	1997.27	112.82	2324.21
stokes	664.27	12772.51	85.47	13522.30	-	-	-	-

4. SUMMARY AND CONCLUSIONS

This dissertation described a multifrontal parallel sparse direct solver's algorithm and implementation using OpenMP tasking for parallelism. The SuiteSparse library environment was used to design ParU. ParU does the symbolic analysis and reads the input matrix using CHOLMOD, AMD, COLAMD, and UMFPACK. For dense kernel calculations, it uses the BLAS library, and some of its algorithms use the C++ STL library. It is optional in ParU to use METIS; however, doing so can speed up factorization by creating a bushy etree for the parallel method. ParU is available under a GNU license and is open source.

The feasibility of a parallel unsymmetric multifrontal algorithm can be proven through ParU's implementation. Although it is developed with OpenMP, the idea and algorithm are generic and can be applied to any upcoming framework. This ParU implementation has numerous challenges. However, these can be alleviated by adopting different frameworks or architectural designs. For instance, optimizing the number of cores operating at each level of parallelism for the best performance is challenging. However, this issue becomes less significant if we use accelerators for large BLAS computations. Additionally, ParU can overlap the communication and processing times while simultaneously calling numerous BLAS functions in parallel.

The program also uses straightforward data structures that are simple to divide and use in a distributed memory setting. The distributed memory implementation of ParU can benefit from the list of live descendants already utilized in the algorithm. In conclusion, ParU is an effective parallel solver, and we think it has a lot of potential for use in various settings.

REFERENCES

- [1] T. A. Davis, “Algorithm 832: UMFPACK V4.3, an unsymmetric-pattern multifrontal method,” *ACM Trans. Math. Softw.*, vol. 30, pp. 196–199, June 2004.
- [2] P. R. Amestoy, M. J. Daydé, and I. S. Duff, “Use of level-3 blas kernels in the solution of full and sparse linear equations,” in *High Performance Computing* (J.-L. Delhaye and E. Gelenbe, eds.), pp. 19–31, Amsterdam: North-Holland, 1989.
- [3] T. A. Davis and I. S. Duff, “An unsymmetric-pattern multifrontal method for sparse LU factorization,” *SIAM J. Matrix Anal. Appl.*, vol. 18, no. 1, pp. 140–158, 1997.
- [4] I. S. Duff and J. K. Reid, “A note on the work involved in no-fill sparse matrix factorization,” *IMA J. Numer. Anal.*, vol. 3, no. 1, pp. 37–40, 1983.
- [5] J. W. H. Liu, “The multifrontal method for sparse matrix solution: theory and practice,” *SIAM Review*, vol. 34, no. 1, pp. 82–109, 1992.
- [6] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK Users’ Guide*. Philadelphia, PA: SIAM, 3rd ed., 1999. <http://www.netlib.org/lapack/lug/>.
- [7] S. Tomov, J. Dongarra, and M. Baboulin, “Towards dense linear algebra for hybrid GPU accelerated manycore systems,” *Parallel Computing*, vol. 36, pp. 232–240, June 2010.
- [8] J. H. Wilkinson and C. Reinsch, eds., *Handbook for Automatic Computation, Volume II: Linear Algebra*. Springer-Verlag, 1971.
- [9] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, “A supernodal approach to sparse partial pivoting,” *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 3, pp. 720–755, 1999.

- [10] J. W. Demmel, J. R. Gilbert, and X. S. Li, “An asynchronous parallel supernodal algorithm for sparse Gaussian elimination,” *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 915–952, 1999.
- [11] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, “A survey of direct methods for sparse linear systems,” *Acta Numerica*, vol. 25, pp. 383–566, 5 2016.
- [12] B. M. Irons, “A frontal solution program for finite element analysis,” *Intl. J. Numer. Methods Eng.*, vol. 2, pp. 5–32, 1970.
- [13] P. Hood, “Frontal solution program for unsymmetric matrices,” *Intl. J. Numer. Methods Eng.*, vol. 10, no. 2, pp. 379–400, 1976.
- [14] M. Yannakakis, “Computing the minimum fill-in is NP-complete,” *SIAM J. Alg. Disc. Meth.*, vol. 2, pp. 77–79, 1981.
- [15] P. R. Amestoy, T. A. Davis, and I. S. Duff, “An approximate minimum degree ordering algorithm,” *SIAM J. Matrix Anal. Appl.*, vol. 17, no. 4, pp. 886–905, 1996.
- [16] P. R. Amestoy, T. A. Davis, and I. S. Duff, “Algorithm 837: AMD, an approximate minimum degree ordering algorithm,” *ACM Trans. Math. Softw.*, vol. 30, pp. 381–388, Sept. 2004.
- [17] 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. Softw.*, vol. 30, pp. 377–380, Sept. 2004.
- [18] G. Birkhoff and A. George, “Elimination by nested dissection,” in *Complexity of Sequential and Parallel Numerical Algorithms* (J. F. Traub, ed.), pp. 221–269, New York: Academic Press, 1973.
- [19] G. Karypis and V. Kumar, “A parallel algorithm for multilevel graph partitioning and sparse matrix ordering,” *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 71–95, 1998.
- [20] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998.

- [21] I. S. Duff and J. K. Reid, “The multifrontal solution of indefinite sparse symmetric linear equations,” *ACM Trans. Math. Softw.*, vol. 9, no. 3, pp. 302–325, 1983.
- [22] P. R. Amestoy, I. S. Duff, S. Pralet, and C. Vömel, “Adapting a parallel sparse direct solver to architectures with clusters of SMPs,” *Parallel Computing*, vol. 29, no. 11–12, pp. 1645 – 1668, 2003.
- [23] T. A. Davis, “A column pre-ordering strategy for the unsymmetric-pattern multifrontal method,” *ACM Trans. Math. Softw.*, vol. 30, pp. 165–195, June 2004.
- [24] D. A. Calahan, “Parallel solution of sparse simultaneous linear equations,” in *Proceedings of the 11th Annual Allerton Conference on Circuits and System Theory*, pp. 729–735, 1973.
- [25] X. S. Li and J. W. Demmel, “SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems,” *ACM Trans. Math. Softw.*, vol. 29, pp. 110–140, June 2003.
- [26] I. S. Duff, “The impact of high-performance computing in the solution of linear systems: trends and problems,” *J. Comput. Appl. Math.*, vol. 123, no. 1-2, pp. 515–530, 2000.
- [27] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, “A set of level-3 basic linear algebra subprograms,” *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.
- [28] K. Goto and R. van de Geijn, “High performance implementation of the level-3 BLAS,” *ACM Trans. Math. Softw.*, vol. 35, pp. 14:1–14:14, July 2008.
- [29] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, “Flame: Formal linear algebra methods environment,” *ACM Trans. Math. Softw.*, vol. 27, pp. 422–455, Dec. 2001.
- [30] T. Mattson, Y. He, and A. Koniges, *The OpenMP Common Core: Making OpenMP Simple Again*. Scientific and Engineering Computation, MIT Press, 2019.
- [31] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

- [32] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, “A column approximate minimum degree ordering algorithm,” *ACM Trans. Math. Softw.*, vol. 30, pp. 353–376, Sept. 2004.
- [33] P. R. Amestoy, T. A. Davis, and I. S. Duff, “Algorithm 837: AMD, an approximate minimum degree ordering algorithm,” *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 381–388, 2004.
- [34] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra, “The Matrix Market: A web resource for test matrix collections,” in *Quality of Numerical Software, Assessment and Enhancement* (R. F. Boisvert, ed.), pp. 125–137, London: Chapman & Hall, 1997. (<http://math.nist.gov/MatrixMarket>).
- [35] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, “Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate,” *ACM Trans. Math. Softw.*, vol. 35, no. 3, pp. 1–14, 2008.
- [36] T. A. Davis and W. W. Hager, “Dynamic supernodes in sparse Cholesky update/downdate and triangular solves,” *ACM Trans. Math. Softw.*, vol. 35, no. 4, pp. 1–23, 2009.
- [37] 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. Softw.*, vol. 30, no. 3, pp. 377–380, 2004.
- [38] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, “A column approximate minimum degree ordering algorithm,” *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 353–376, 2004.
- [39] 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. Softw.*, vol. 16, pp. 1–17, 1990.
- [40] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, “The SuiteSparse matrix collection website interface,” *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019.
- [41] T. A. Davis and E. Palamadai Natarajan, “Algorithm 907: KLU, a direct sparse solver for circuit simulation problems,” *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, Sept. 2010.

APPENDIX A

DETAILED PERFORMANCE METRICS AND COMPUTING ENVIRONMENT

The topology of our machine is shown in Figure A.2.

Hotspots ©		Analysis Configuration	Collector/Callee	Bottom-up	Caller/Callee	Top-down Tree	Flame Graph	Platform	Function (Full)
		Function	CPU Time: Total	CPU Time: Self	Module				
clone	54.5%	0s	libc.so.6	clone					
__kmp_launch_thread	54.5%	0s	libomp5.so	__kmp_launch_thread					
[Loop at line 6262 in __kmp_launch_thread]	54.5%	202.587s	libomp5.so	[Loop at line 6262 in __kmp_launch_thread]					
[Loop at line 530 in _INTERNALfae7a54c::OpenMP w	54.5%	0s	libomp5.so	[Loop at line 530 in _INTERNALfae7a54c::OpenMP w					
[Loop at line 530 in _INTERNALfae7a54c::OpenMP w	54.5%	0s	libomp5.so	[Loop at line 530 in _INTERNALfae7a54c::OpenMP w					
_INTERNALfae7a54c::OpenMP worker]	54.5%	0s	libomp5.so	_INTERNALfae7a54c::OpenMP worker]					
start_thread	54.5%	0s	libpthread.so.0	start_thread					
_start	45.2%	0s	x_paru_demo	_start					
main	45.2%	0s	x_paru_demo	main					
paru_front	42.6%	0.323s	libparu.so.0	paru_front					
ParU_Factorize	42.3%	0s	libparu.so.0	ParU_Factorize					
[Loop@0xc4e0 in paru_exec_tasks_seq]	41.4%	0s	libparu.so.0	[Loop@0xc4e0 in paru_exec_tasks_seq]					
paru_exec_tasks_seq	41.4%	0s	libparu.so.0	paru_exec_tasks_seq					
[Loop@0xc49c in paru_exec_tasks_seq]	41.4%	0s	libparu.so.0	[Loop@0xc49c in paru_exec_tasks_seq]					
[OpenMP fork]	38.5%	4.130s	libomp5.so	[OpenMP fork]					
[OpenMP dispatcher]	37.4%	120.248s	libomp5.so	[OpenMP dispatcher]					
paru_tasked_dgemm	29.9%	0s	libparu.so.0	paru_tasked_dgemm					
dgemm_	29.9%	0.020s	libmkl_intel_lp64.so.2	dgemm_					
cbias_dgemm	21.1%	0s	libmkl_intel_lp64.so.2	cbias_dgemm					
[Loop@0xea10 in paru_factorize_full_summed]	21.1%	0s	libparu.so.0	[Loop@0xea10 in paru_factorize_full_summed]					
paru_factorize_full_summed	21.1%	0s	libparu.so.0	paru_factorize_full_summed					
paru_dgemm	11.8%	0s	libparu.so.0	paru_dgemm					
[Loop@0x12e61 in paru_prior_assemble]	7.6%	0.008s	libparu.so.0	[Loop@0x12e61 in paru_prior_assemble]					
paru_prior_assemble	7.6%	0s	libparu.so.0	paru_prior_assemble					
paru_assemble_all	7.3%	0s	libparu.so.0	paru_assemble_all					
[Loop@0x129d0 in _Z17paru_assemble_allIRSt6vector	5.3%	0.064s	libparu.so.0	[Loop@0x129d0 in _Z17paru_assemble_allIRSt6vector					
_Z17paru_assemble_allIRSt6vectorISaIEEP9paru_wor	5.3%	0s	libparu.so.0	_Z17paru_assemble_allIRSt6vectorISaIEEP9paru_wor					
[Loop@0x12a70 in _Z17paru_assemble_allIRSt6vecto	5.3%	20.198s	libparu.so.0	[Loop@0x12a70 in _Z17paru_assemble_allIRSt6vecto					
paru_panel_factorize]	2.2%	0.020s	libparu.so.0	paru_panel_factorize]					
paru_panel_factorize	2.2%	0s	libparu.so.0	paru_panel_factorize					
cholmod_l_read_matrix	1.7%	0s	libcholmod.so.3	cholmod_l_read_matrix					
[Loop@0xe610 in paru_panel_factorize]	1.6%	6.241s	libparu.so.0	[Loop@0xe610 in paru_panel_factorize]					
read_triplet	1.5%	0s	libcholmod.so.3	read_triplet					
[Loop@0xb5be6 in read_triplet]	1.5%	4.806s	libcholmod.so.3	[Loop@0xb5be6 in read_triplet]					
ParU_Factorize.extracted.32	1.1%	0s	libparu.so.0	ParU_Factorize.extracted.32					
[Loop@0xc5b0 in paru_exec_tasks]	1.1%	0s	libparu.so.0	[Loop@0xc5b0 in paru_exec_tasks]					
paru_exec_tasks	1.1%	0s	libparu.so.0	paru_exec_tasks					
[Loop@0xc561 in paru_exec_tasks]	1.1%	0s	libparu.so.0	[Loop@0xc561 in paru_exec_tasks]					
ParU_Analyze	1.1%	0.032s	libparu.so.0	ParU_Analyze					
umfpack_dl_paru_symbolic	1.0%	0s	libumfpack.so.5	umfpack_dl_paru_symbolic					
symbolic_analyze	1.0%	0s	libumfpack.so.5	symbolic_analyze					

Figure A.1: Intel Vtune graph for ParU



Figure A.2: Topology of our machine