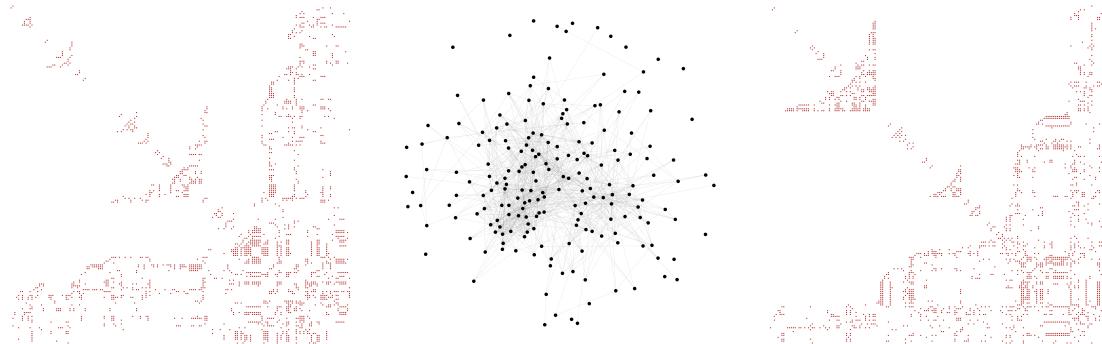


DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING
Fall Semester 2025

Comprehensive Evaluation of Sparse Symmetric Matrix Permutation Methods

Master Thesis



Ritvik Ranjan
rranjan@ethz.ch

October 5, 2025

Advisors: Vincent Maillou, vmaillou@iis.ee.ethz.ch
Dr. Alexandros Nikolaos Ziogas, alziogas@iis.ee.ethz.ch
Professor: Prof. Dr. Mathieu Luisier, leader@iis.ee.ethz.ch

Abstract

Sparse symmetric linear systems arise across computational science and engineering. Solving these systems efficiently via direct methods depends critically on matrix reordering to minimize fill-in during factorization. Finding optimal reorderings is NP-complete, necessitating good heuristic approaches. This thesis presents a comprehensive evaluation of sparse matrix reordering algorithms, comparing classical methods, graph partitioning approaches, and hypergraph-based techniques across diverse matrix structures. We assess multiple performance metrics including fill-in reduction, execution time, memory usage, and parallelization potential. Our benchmarking reveals that advanced methods achieve better fill-in reduction compared to classical algorithms, though at higher computational cost. We implemented a few enhancements, including a Nested Dissection variant with improved coarsening and a GPU-accelerated RCM implementation, demonstrating significant performance improvements. The results establish some trade-offs between ordering quality and computational overhead, providing guidance for algorithm selection and future research directions in sparse matrix reordering.

On the cover: Reorderings and the original graph of the Jazz musicians (1912-1940) collaboration network.

Acknowledgments

First and foremost, I would like to thank Prof. Dr. Mathieu Luisier for providing me with the opportunity to work in the Nano-TCAD group and on this fascinating topic.

I would like to thank Vincent Maillou for providing me the opportunity to work on this project and for his friendly and helpful supervision and the patience and time put into teaching me different aspects of it. I express my sincere thanks to Dr. Alexandros Nikolaos Ziogas for being supportive and guiding me through the process.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

- I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies¹.
- I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

Comprehensive Evaluation of Sparse Symmetric Matrix Permutation Methods

Authored by:

If the work was compiled in a group, the names of all authors are required.

Last name(s):

Ranjan

First name(s):

Ritvik

With my signature I confirm the following:

- I have adhered to the rules set out in the [Citation Guidelines](#).
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date

Zürich, 06.10.25

Signature(s)

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

¹ For further information please consult the ETH Zurich websites, e.g. <https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html> and <https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html> (subject to change).

Contents

| | |
|---|-----------|
| 1. Introduction | 1 |
| 2. Background | 3 |
| 2.1. Elimination Trees and Symbolic Factorization | 3 |
| 2.1.1. Mathematical Foundations and Problem Formulation | 3 |
| 2.1.2. Structural Properties | 5 |
| 2.1.3. Algorithmic Construction of Elimination Trees | 5 |
| 2.1.4. Symbolic Factorization | 6 |
| 2.2. Minimum fill-in and NP-completeness | 8 |
| 2.3. Heuristics classification | 8 |
| 2.3.1. Bandwidth Minimization | 9 |
| 2.3.2. Minimum Degree | 9 |
| 2.3.3. Nested Dissection | 10 |
| 2.3.4. Hypergraph partitioning and other methods | 11 |
| 3. Implementation and Optimizations | 12 |
| 3.1. SuiteSparse implementation of RCM and Minimum Degree | 12 |
| 3.1.1. RCM Algorithm Overview | 12 |
| 3.1.2. AMD Algorithm Overview | 12 |
| 3.2. Nested Dissection using METIS and SCOTCH | 14 |
| 3.3. New Coarsening Approaches in Nested Dissection | 16 |
| 3.4. Hypergraph Based Ordering | 19 |
| 3.5. GPU Implementation of RCM | 22 |
| 4. Results | 24 |
| 4.1. Evaluation setup | 24 |
| 4.1.1. Hardware Infrastructure | 24 |
| 4.2. Matrices dataset | 25 |
| 4.3. Methodology | 26 |
| 4.3.1. Performance Metrics | 27 |

Contents

| | |
|--|-----------|
| 4.4. Evaluation | 28 |
| 4.4.1. GPU-accelerated RCM | 38 |
| 4.5. Current limitations | 39 |
| 5. Other Approaches | 40 |
| 5.1. Graph Reinforcement Learning for Reordering | 40 |
| 5.2. GPU Accelerated Nested Dissection | 41 |
| 6. Conclusion and Future Work | 43 |
| A. Experimental Results | 45 |
| B. Task Description | 51 |
| List of Figures | 52 |
| List of Tables | 54 |
| Bibliography | 56 |

Chapter 1

Introduction

Sparse linear systems of the form $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is sparse and symmetric positive definite, are ubiquitous in modern scientific computing. These systems arise naturally across virtually every domain of computational science and engineering, including quantum chemistry, computer graphics, computational fluid dynamics, power networks, machine learning, and optimization. Formally, one may state matrix A is considered sparse when the number of nonzero entries $\text{nnz}(A)$ satisfies $\text{nnz}(A) = O(n)$ rather than $O(n^2)$ as in dense matrices, although a better practical definition is a matrix is sparse if it is advantageous to exploit its zero entries [1].

This sparsity typically arises from the discretization of partial differential equations (PDEs)[2] using finite difference, finite element, or finite volume methods. When continuous problems described by partial differential equations are discretized using methods such as finite differences, finite elements, or finite volumes, the resulting algebraic equations connect only neighboring (or some specific) grid points or elements. Conceptually, sparsity corresponds to systems with few pairwise interactions. For instance, in a three-dimensional finite element mesh, each node typically connects to only a small neighborhood of adjacent nodes, regardless of the total problem size, resulting in matrices where the number of non-zero elements is roughly equal to the number of rows or columns rather than being proportional to n^2 .

As we see in Fig. 1.1, the Cholesky factorization of a sparse matrix often results in a denser matrix due to fill-in, where zero entries in the original matrix become non-zero in the factorized form. This fill-in can significantly increase both memory usage and computational time for solving the system. The amount of fill-in depends on the order in which variables are eliminated during factorization, which is determined by the ordering, or permutation, of the matrix rows and columns. The permutations serve as a preprocessing step that can reduce fill-in largely. The basic principle involves reordering the rows and columns of the matrix A to obtain a permuted system $PAP^T\hat{x} = Pb$, where P is a permutation matrix, such that the reordered matrix exhibits superior properties for factorization.

In this thesis, we focus exclusively on the permutation problem for sparse symmetric

1. Introduction

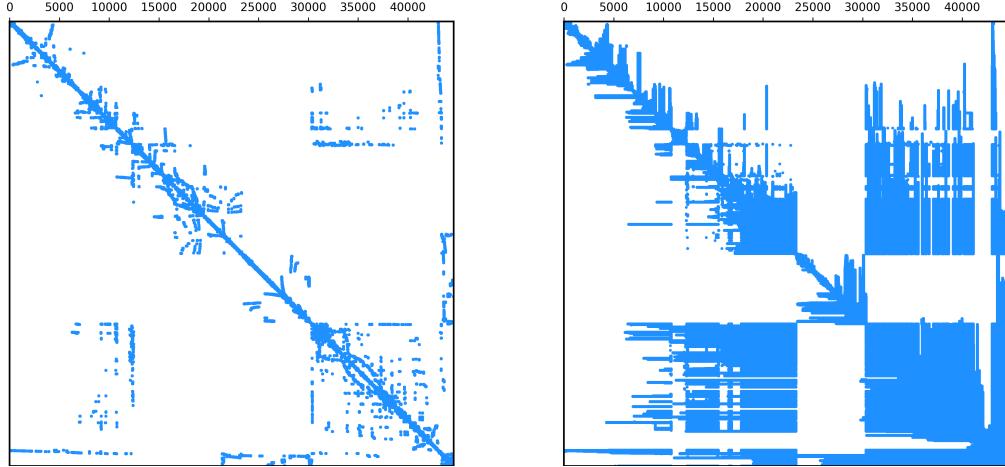


Figure 1.1.: Sparsity pattern of a matrix from structural engineering bcsstk32 (left) and $L + L^T$ (right) where L is the Cholesky factor of bcsstk32.

matrices. We explore existing methods for sparse symmetric matrix reordering, investigate potential optimizations and parallelization strategies, and evaluate the performance of different reordering methods across a variety of matrices, assessing their effectiveness in terms of fill-in reduction, memory usage and parallelism. In chapter 2, we provide the necessary theoretical background on sparse matrix factorization and graph representations. Chapter 3 details the implementation of various reordering algorithms, while chapter 4 presents an overview of our experimental evaluation. For full performance results, please refer to the appendix A.

Chapter 2

Background

Before we look into the algorithms and theory behind reordering techniques, we first take a dive into the factorization process that produces further non-zeroes, which in this and the entire thesis's context, are called fill-in. It may be trivial but important to note, that if some numerical arithmetic results in a zero in the factored matrix, it is usually still considered as a fill-in.

Suppose the set of n equations that are to be solved are

$$Ax = b \quad (2.1)$$

where A is a sparse matrix, x is the vector of unknowns. The sparsity, or the number of non-zero entries in A determines the fill-in of its Cholesky factor when it is employed to solve the aforementioned set of equations. Suppose the Cholesky factorization of A is given by LL^T , where L is a lower triangular matrix (with a positive diagonal) and L^T is the transpose of L . The efficiency of solving this set of equations is dependent on the number of non-zero entries in L . It has been shown that we can factor and solve the set of equations in space proportional to $\sum_j d_j$ and time complexity $\sum_j d_j^2$, where d_j is the number of non-zeroes in the j th column of L [3].

2.1. Elimination Trees and Symbolic Factorization

2.1.1. Mathematical Foundations and Problem Formulation

Consider a sparse symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$. The Cholesky factorization decomposes A into the form $A = LL^T$, where L is a lower triangular matrix. While A may contain relatively few nonzero entries, the factor L typically exhibits significantly more nonzeros due to a phenomenon known as fill-in. This fill-in occurs because the elimination process creates new nonzero entries at positions that were originally zero in A .

The sparsity pattern of A is defined as the set $\text{pattern}(A) = \{(i, j) : A_{ij} \neq 0\}$. Similarly, $\text{pattern}(L)$ denotes the sparsity pattern of the Cholesky factor. The fundamental

2. Background

observation is that $\text{pattern}(L)$ can be completely determined from $\text{pattern}(A)$ using purely structural analysis, without requiring any numerical computation. This structural predictability forms the theoretical foundation for elimination trees and symbolic factorization.

Every sparse symmetric matrix A can be naturally represented as an undirected graph $G(A) = (V, E)$, where the vertex set $V = \{1, 2, \dots, n\}$ corresponds to the rows and columns of A , and the edge set E contains an edge (i, j) if and only if $A_{ij} \neq 0$ for $i \neq j$. This graph representation transforms matrix operations into graph-theoretic problems, helping us to reformulate a lot of the concepts in terms of graph properties.

The elimination process on matrix A corresponds to a sequence of vertex eliminations on graph $G(A)$. When vertex k is eliminated, all pairs of neighbors of k become connected by edges if they were not already connected. This edge addition process continues throughout the elimination sequence, producing what is known as the filled graph $G^+(A)$. The filled graph contains all edges that exist at any point during the elimination process, and its structure completely determines the sparsity pattern of the Cholesky factor L .

The elimination tree provides a compact representation of the structural relationships inherent in sparse matrix factorization. This tree structure captures the essential dependencies between different stages of the elimination process.

Definition 2.1 (Elimination Tree). Let A be a symmetric positive definite matrix with Cholesky factorization $A = LL^T$. The elimination tree $T(A) = (V, E_T)$ is a directed tree defined by the parent function $\text{parent} : V \rightarrow V \cup \{\emptyset\}$, where:

$$\text{parent}(j) = \min\{i > j : L_{ij} \neq 0\} \quad (2.2)$$

If no such index i exists, then $\text{parent}(j) = \emptyset$ and j is a root of the tree. The directed edges of the tree are given by $E_T = \{(j, \text{parent}(j)) : \text{parent}(j) \neq \emptyset\}$.

This definition establishes a precise correspondence between the numerical structure of the Cholesky factor and the combinatorial structure of a directed tree. Each node j in the tree corresponds to column j of the matrix, and the parent relationship encodes which column will first modify column j during the factorization process.

Definition 2.2 (Ancestor Relationship). In the elimination tree $T(A)$, node i is an ancestor of node j (denoted $i \succ j$) if there exists a directed path from j to i in the tree. Node i is a proper ancestor if $i \succ j$ and $i \neq j$.

The ancestor relationship in the elimination tree is closely tied to the structure of the Cholesky factor, as formalized in the following theorem.

Theorem 2.1 (Characterization Theorem). *Let $T(A)$ be the elimination tree of symmetric positive definite matrix A with Cholesky factor L . Then for any indices $i > j$: $L_{ij} \neq 0$ if and only if $i \succ j$ in $T(A)$.*

This theorem establishes that the elimination tree completely characterizes the sparsity pattern of the Cholesky factor. The nonzero entries in column j of L correspond exactly to

2. Background

the ancestors of node j in the elimination tree. This remarkable result shows that purely combinatorial information (tree ancestry) determines numerical structural properties (matrix sparsity patterns).

2.1.2. Structural Properties

Fill-in Characterization Through Path Analysis

Understanding when fill-in occurs during elimination requires analyzing connectivity patterns in the original matrix graph. The concept of fill paths provides a precise characterization of this phenomenon.

Definition 2.3 (Fill Path). Let $G(A) = (V, E)$ be the graph of matrix A . A fill path from vertex i to vertex j (where $i < j$) is a path $P = (i = v_0, v_1, v_2, \dots, v_k = j)$ in $G(A)$ such that all intermediate vertices satisfy $v_l < \min(i, j)$ for $1 \leq l \leq k - 1$.

The fill path condition ensures that all intermediate vertices on the path are eliminated before either endpoint, which is precisely the condition under which the elimination process will create a direct connection between i and j .

Theorem 2.2 (Fill Path Characterization). *An edge (i, j) with $i < j$ belongs to the filled graph $G^+(A)$ if and only if there exists a fill path from i to j in the original graph $G(A)$.*

This theorem provides the theoretical foundation for predicting fill-in patterns. It shows that structural properties of the original graph completely determine which new edges will be created during elimination, independent of the specific numerical values in the matrix.

The elimination tree possesses several important structural properties that make it amenable to efficient algorithmic manipulation.

Theorem 2.3 (Postorder Property). *In any elimination tree $T(A)$, if $\text{parent}(j) = i$, then all nodes in the subtree rooted at j have indices less than i . Furthermore, the elimination tree is uniquely determined by the sparsity pattern of A , independent of the specific elimination ordering used (provided the ordering respects the natural ordering of indices).*

This property implies that elimination trees can be processed using postorder traversal, where children are visited before their parents. This traversal order naturally reflects the dependencies in the elimination process—a column can only be processed after all columns in its subtree have been completed.

2.1.3. Algorithmic Construction of Elimination Trees

The most straightforward approach to constructing elimination trees directly implements the definition by examining the sparsity structure of the matrix.

This algorithm requires $O(|A|)$ time, where $|A|$ denotes the number of nonzero entries in matrix A . The space complexity is $O(n)$ for storing the parent array. While simple

2. Background

Algorithm 1: Direct Construction Method

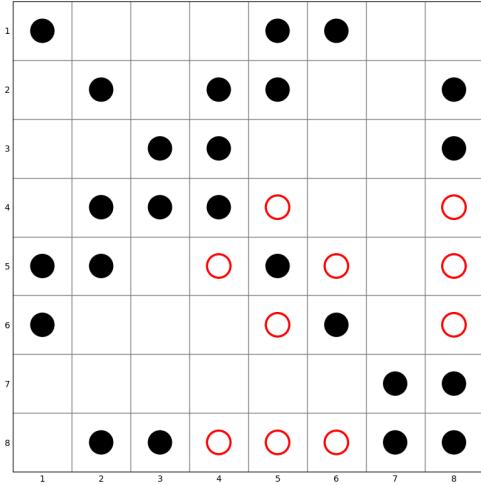
Input :Symmetric positive definite matrix A with sparsity pattern
Output:Parent array representing elimination tree $T(A)$

```

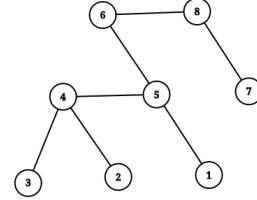
1 Initialize parent[1...n] ← 0;                                // 0 represents null parent
2 for  $j = 1$  to  $n$  do
3   parent[j] ← 0;
4   foreach  $i \in \{k : A[j,k] \neq 0 \text{ and } k > j\}$  do
5     if parent[j] = 0 or  $i < \text{parent}[j]$  then
6       parent[j] ← i;
7     end
8   end
9 end
10 return parent array

```

to understand and implement, this direct approach does not exploit the structural properties of elimination trees for potential efficiency improvements.



(a) Sparse matrix structure



(b) Corresponding elimination tree

Figure 2.1.: Illustration of elimination tree construction for the given sparse symmetric matrix. The sparsity is denoted by filled black circles and the fill-in induced is denoted by hollow red circles.

2.1.4. Symbolic Factorization

Symbolic factorization determines the sparsity pattern of the Cholesky factor L from the sparsity pattern of matrix A using the elimination tree. It does not involve any numerical

2. Background

computations and relies solely on the structure of the original matrix.

The key insight is that the sparsity structure of each column in L can be computed by analyzing how structural information propagates through the elimination tree. Each column inherits structure from two sources: the original matrix A and the previously computed columns corresponding to its children in the elimination tree.

Definition 2.4 (Column Structure Inheritance). For column j of the Cholesky factor L , the nonzero pattern is determined by:

- Direct inheritance: All nonzero entries from column j of the original matrix A that lie on or below the diagonal
- Indirect inheritance: All entries from the union of nonzero patterns of columns corresponding to children of j in the elimination tree, restricted to rows numbered greater than j

This inheritance pattern reflects the computational dependencies during numerical factorization—column j can only be processed after all its children in the elimination tree have been completed.

Algorithm 2: Elimination Tree Symbolic Factorization

Input :Sparsity pattern of matrix A , elimination tree $T(A)$
Output:Sparsity pattern of Cholesky factor L

```

1 Initialize column_structure[1...n] as empty sets;
2 for  $j = 1$  to  $n$ ;                                // in postorder traversal of  $T(A)$ 
3   do
4     // Direct inheritance from original matrix
5     column_structure[j]  $\leftarrow \{i \geq j : A[i,j] \neq 0\}$ ;
6     // Indirect inheritance from children in elimination tree
7     foreach child  $c$  of  $j$  in  $T(A)$  do
8       | column_structure[j]  $\leftarrow$  column_structure[j]  $\cup \{i \in \text{column\_structure}[c] :$ 
9       |    $i > j\}$ ;
10      end
11      // Record structure for column  $j$  of  $L$ 
12      pattern( $L[:,j]$ )  $\leftarrow$  column_structure[j];
13    end
14  return pattern( $L$ )

```

The postorder traversal ensures that each column is processed only after all its descendants in the elimination tree have been completed, respecting the computational dependencies inherent in the factorization process.

2. Background

2.2. Minimum fill-in and NP-completeness

When performing Cholesky factorization on a sparse symmetric matrix, we want to minimize fill-in - the new nonzero entries that appear during elimination. This translates to a graph theory problem: given a graph representing the matrix's sparsity pattern, find an elimination order that creates the fewest new edges. Yannakakis [4] proved that this optimization problem is NP-complete, meaning no polynomial-time algorithm can solve it optimally in general (unless P = NP).

The key insight connecting linear algebra to graph theory is that chordal graphs are essential in proving this. A graph is chordal if every cycle of length ≥ 4 has a chord (an edge connecting non-consecutive vertices). The fundamental theorem states that a graph has perfect elimination (zero fill-in) if and only if it is chordal, and any elimination process on any graph produces a chordal result. Therefore, the minimum fill-in problem becomes: What is the smallest number of edges we must add to make our graph chordal?

The proof uses a three-step reduction to prove NP-completeness. First, we consider bipartite graphs and their special case, chain graphs. A bipartite graph has vertices split into two independent sets P and Q , and it is a chain graph when neighborhoods form a nested sequence: $\Gamma(v_1) \subseteq \Gamma(v_2) \subseteq \dots \subseteq \Gamma(v_k)$. The key property is that a bipartite graph is a chain graph if and only if it contains no pair of independent edges.

Second, for any bipartite graph $G = (P, Q, E)$, we construct $C(G)$ by making P into a complete clique (all vertices in P connected), making Q into a complete clique (all vertices in Q connected), and keeping original edges between P and Q . The crucial lemma states that $C(G)$ is chordal if and only if G is a chain graph. This works because if G has independent edges, they create a 4-cycle in $C(G)$ with no chord. Conversely, if G is a chain graph, $C(G)$ has a perfect elimination order.

Third, the reduction uses the Optimal Linear Arrangement Problem, which asks: given a graph, arrange vertices on a line to minimize the sum of distances between adjacent vertices. This problem is known to be NP-complete. Yannakakis constructs an transformation where, given graph G with n vertices and m edges, he creates bipartite graph G' where part P has one vertex for each vertex in G , and part Q has an elaborate gadget structure with $2m + n^2 - d(v)$ vertices, with connections that encode the linear arrangement constraints. The mathematical relationship is:

$$\text{Minimum fill-in of } G' = \text{Optimal arrangement cost of } G + \frac{n^2(n-1)}{2} - 2m \quad (2.3)$$

This result has practical implications that no perfect algorithm exists, as any algorithm guaranteeing optimal fill-in will require exponential time in the worst case. This explains why practical sparse matrix reordering uses heuristics like minimum degree ordering, nested dissection and such.

2.3. Heuristics classification

There are several heuristics that have been proposed to reduce the fill-in developed over the years. These heuristics can be broadly classified in the following categories:

2. Background

Bandwidth minimization, minimum degree (and its variants), nested dissection, banded structure methods using hypergraphs, and recently, machine learning approaches.

2.3.1. Bandwidth Minimization

Bandwidth minimization refers to the problem of permuting the rows and columns of a matrix such that the non-zero entries are as close to the diagonal as possible. Gaussian elimination can be performed in $O(nb^2)$ time on matrices of dimension n and bandwidth b , which is faster than the forward $O(n^3)$ algorithm when b is smaller than n [5]. Additionally, this problem is NP-complete [6], but several heuristics have been developed to approximate the optimal solution.

The Cuthill-McKee algorithm [7] employs breadth-first search to reduce matrix bandwidth by generating level structures. However, it has computational limitations and may not achieve optimal bandwidth reduction. The Reverse Cuthill-McKee algorithm [8] addresses these issues by reversing the ordering, while the GPS algorithm by Gibbs et al. provides an alternative level structure approach.

2.3.2. Minimum Degree

The minimum degree algorithm [9] determines the order in which to eliminate variables (or pivot) during Gaussian elimination to minimize fill-in (creation of new non-zero entries) in sparse matrices. At each step, it chooses the node with the minimum degree (fewest connections) for elimination.

The minimum degree algorithm operates on the adjacency graph of the sparse matrix. It begins by initializing the graph, where each vertex represents a variable. At each iteration, the algorithm selects the vertex with the smallest degree (i.e., the fewest neighbors), which corresponds to the variable whose elimination is expected to introduce the least fill-in. Upon elimination, the chosen vertex is removed from the graph, and all its neighbors are connected to each other, forming a clique to preserve the matrix structure. The degrees of the affected vertices are then updated to reflect the new connections. This process is repeated until all vertices have been eliminated. The resulting elimination order directly determines the pivot sequence used during matrix factorization.

Variants of the minimum degree algorithm include:

1. Multiple Minimum Degree (MMD) [10]

When multiple nodes have the same minimum degree, uses additional tie-breaking rules. Often selects based on secondary criteria like external degree or node numbering.

2. Approximate Minimum Degree (AMD) [11]

Uses approximations to avoid expensive exact degree calculations. Employs concepts like "supervariables" and "element absorption". Much faster than exact minimum degree while maintaining similar fill-in quality.

2. Background

3. Constrained Minimum Degree

Incorporates additional constraints (like maintaining bandwidth). Balances minimum degree objective with other structural requirements.

2.3.3. Nested Dissection

Nested dissection is a divide and conquer heuristic for the solution of sparse symmetric systems of linear equations based on graph partitioning. Nested dissection can be viewed as a recursive divide-and-conquer algorithm on an undirected graph; it uses separators in the graph, which are small sets of vertices whose removal divides the graph approximately in half [12].

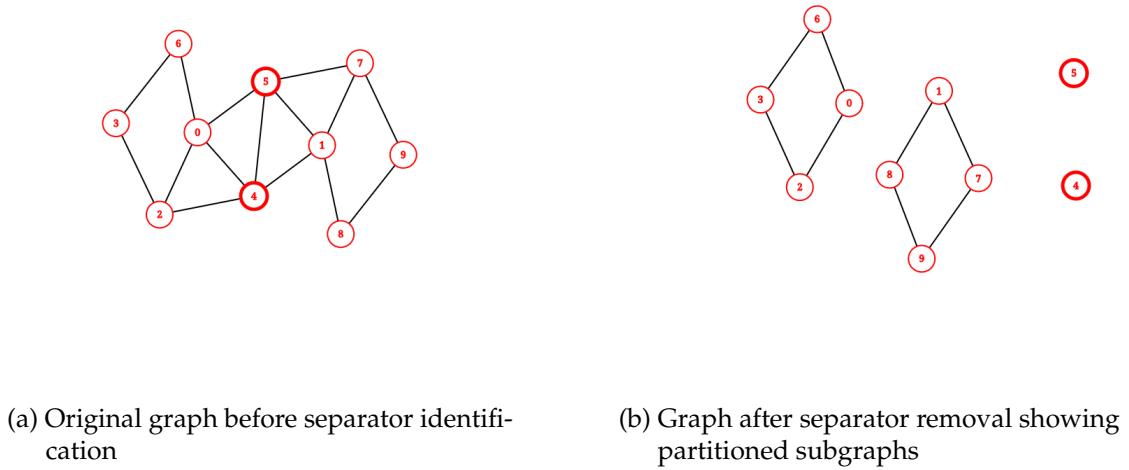


Figure 2.2.: Separator identification process. The separator vertices divide the graph into approximately equal subgraphs

Nested dissection consists of the following steps: Form an undirected graph in which the vertices represent rows and columns of the system of linear equations, and an edge represents a nonzero entry in the sparse matrix representing the system. Recursively partition the graph into subgraphs using separators, small subsets of vertices the removal of which allows the graph to be partitioned into subgraphs with at most a constant fraction of the number of vertices. Perform Cholesky decomposition (a variant of Gaussian elimination for symmetric matrices), ordering the elimination of the variables by the recursive structure of the partition: each of the two subgraphs formed by removing the separator is eliminated first, and then the separator vertices are eliminated [13].

All the sequential algorithms for determining the elimination ordering of a graph G can be described by the following general algorithm: Generate the tree of separators for G and perform a tree traversal on the separator tree to order the vertices, where this

2. Background

traversal must visit a node before any of its parents.

2.3.4. Hypergraph partitioning and other methods

In recent years, hypergraph partitioning methods [14] have been proposed to reduce fill-in during sparse matrix factorization. Hypergraphs generalize graphs by allowing edges (hyperedges) to connect more than two vertices, making them well-suited for modeling complex relationships in sparse matrices. In this paper we explore another Hypergraph based reordering method,

Other methods include machine learning approaches that learn optimal ordering strategies using reinforcement learning. One such approach is Alpha Elimination [15] which uses Convolutional Neural Networks (CNNs) and reinforcement learning to predict the next node to eliminate based on the current state of the graph. We have explored a more flexible architecture but fails with scalability, which we discuss in Chapter 5.

Chapter 3

Implementation and Optimizations

This chapter describes both the existing state-of-the-art implementations that form the baseline for comparison (Sections 3.1 and 3.2), as well as the new algorithms and implementations developed in this thesis (Sections 3.3, 3.4, and 3.5).

3.1. SuiteSparse implementation of RCM and Minimum Degree

3.1.1. RCM Algorithm Overview

The Reverse Cuthill-McKee (RCM) algorithm in SuiteSparse [16, 17] provides bandwidth reduction for symmetric sparse matrices through a breadth-first search strategy combined with degree-based ordering heuristics. The implementation processes disconnected components separately and employs pseudo-peripheral vertex selection to minimize profile and bandwidth.

The SuiteSparse RCM implementation incorporates several refinements over the basic algorithm. The pseudo-peripheral vertex selection uses multiple BFS traversals to identify vertices that are approximately diametrically opposite, which typically results in better bandwidth reduction than arbitrary starting points. The degree-based sorting of neighbors during BFS traversal helps create a more systematic ordering that tends to group low-degree vertices together, further improving the resulting bandwidth.

The algorithm's effectiveness stems from its ability to produce orderings where vertices with similar connectivity patterns are placed close together in the permutation. This locality property translates directly into reduced bandwidth and improved cache performance during matrix operations, making RCM particularly valuable for iterative solvers and direct factorization methods that benefit from band structure preservation.

3.1.2. AMD Algorithm Overview

The Approximate Minimum Degree (AMD) algorithm implemented in SuiteSparse [16, 17, 18] follows a refined elimination-based approach that balances computational

3. Implementation and Optimizations

Algorithm 3: SuiteSparse RCM Algorithm

```

input :Symmetric matrix  $A$  ( $n \times n$ )
output:Permutation  $P$  for bandwidth reduction

1 Initialize;;
2 Compute degree[ $i$ ] =  $|\text{adj}(i)|$  for all vertices  $i$ ;
3 Find connected components  $R_1, R_2, \dots, R_k$  using DFS;
4 Initialize visited[ $i$ ] = false for all  $i$ ;
5 Set perm_index = 0;

6 Process each component;;
7 foreach connected component  $R_j$  do
8   Find pseudo-peripheral start vertex;;
9   start = FindPseudoPeripheral( $R_j$ );
   // Select vertex with minimum degree among maximum-distance
   // vertices
10  Cuthill-McKee BFS traversal;;
11  Initialize queue  $Q = \{\text{start}\}$ ;
12  Set visited[start] = true;
13  Initialize CM_order = [start];
14  while  $Q \neq \emptyset$  do
15     $u = \text{dequeue}(Q)$ ;
16    neighbors = sort(unvisited_adj( $u$ ), by_degreeAscending);
17    foreach  $v \in \text{neighbors}$  do
18      if  $\neg\text{visited}[v]$  then
19        Set visited[v] = true;
20        enqueue( $Q, v$ );
21        Append  $v$  to CM_order;
22      end
23    end
24  end
25  Apply reverse ordering (RCM);;
26  for  $i = 0$  to  $|\text{CM\_order}| - 1$  do
27    |  $P[\text{perm\_index} + i] = \text{CM\_order}[|\text{CM\_order}| - 1 - i]$ ;
28  end
29  perm_index  $\leftarrow$  perm_index +  $|\text{CM\_order}|$ ;
30 end
31 return  $P$ ;

```

3. Implementation and Optimizations

efficiency with fill-in minimization. The algorithm operates on a quotient graph representation and incorporates several key optimizations including aggressive absorption, approximate degree updates, and dense row detection.

The main efficiency gain in SuiteSparse's AMD implementation lies in its aggressive absorption strategy and approximate degree computation. The aggressive absorption phase identifies elements that can be completely absorbed into the current pivot, reducing the size of the quotient graph and improving cache locality. The approximate degree updates provide a computationally efficient method to maintain ordering decisions without exact degree computation, which becomes prohibitively expensive as elimination progresses.

The dense row detection mechanism addresses a common pathology in minimum degree algorithms where elimination of high-degree vertices can lead to excessive fill-in. When the algorithm detects that a newly formed element exceeds a threshold based on the matrix size, it defers elimination of the associated variables, effectively implementing a hybrid strategy that combines minimum degree with nested dissection principles.

3.2. Nested Dissection using METIS and SCOTCH

Nested dissection represents a divide-and-conquer approach to matrix ordering that recursively partitions the graph using small vertex separators, ordering the separated components before the separator vertices. METIS and SCOTCH implement sophisticated multilevel nested dissection algorithms that combine graph coarsening, separator finding, and refinement techniques to produce high-quality orderings for large sparse matrices.

The multilevel nested dissection approach in METIS provides superior ordering quality compared to single-level methods by operating at multiple scales. The coarsening phase creates a hierarchy of increasingly smaller graphs while preserving essential structural properties through heavy edge matching. This matching strategy prioritizes edges with large weights, which in the context of matrix ordering typically correspond to strong structural connections that should be preserved during coarsening.

The separator computation on the coarsest level benefits from reduced problem size while maintaining global structural awareness. The refinement phase during projection ensures that separators remain high-quality as they are mapped back to finer graph levels. The recursive application of this process creates a natural hierarchy where large components are isolated first, followed by progressively smaller substructures, resulting in elimination orderings with excellent fill-in characteristics for sparse direct solvers.

3. Implementation and Optimizations

Algorithm 4: SuiteSparse AMD Algorithm

input :Symmetric matrix A ($n \times n$), Control parameters
output:Permutation P , Info statistics

```

1 Initialize::;
2 Convert  $A$  to  $A + A^T$  pattern if unsymmetric;
3 Build quotient graph  $G = (V, E)$  from  $A$ ;
4 Initialize degree lists Head[ $d$ ] for  $d = 0$  to  $n$ ;
5 Set degree[ $v$ ] =  $|\text{adj}(v)|$  for all vertices  $v$ ;
6 Place each vertex in appropriate degree list;

7 Main elimination loop::;
8 for  $k = 1$  to  $n$  do
9   Select pivot::;
10  Find minimum degree  $d$  with non-empty Head[ $d$ ];
11  Select pivot  $p$  from Head[ $d$ ];
12  Remove  $p$  from degree list;
13  Set  $P[k] = p$  (add to elimination ordering);
14  Element absorption::;
15  foreach element  $e$  adjacent to  $p$  do
16    if  $|L_e \cap L_p| = |L_e|$  then
17      | Absorb  $e$  into  $p$  (aggressive absorption);
18    end
19  end
20  Form new element  $e_p$ ::;
21   $L_{e_p} = \text{adj}(p) \setminus \{\text{absorbed elements}\}$ ;
22  Mark  $e_p$  as new element;
23  Update degrees (approximate)::;
24  foreach uneliminated vertex  $v \in L_{e_p}$  do
25    external_degree[ $v$ ] =  $|\text{adj}(v) \cap \text{uneliminated}|$ ;
26    bound =  $|L_{e_p}| - |L_{e_p} \cap \text{adj}(v)|$ ;
27    degree[ $v$ ]  $\approx$  external_degree[ $v$ ] + bound;
28    Move  $v$  to new degree list;
29  end
30  Dense row detection::;
31  if  $|L_{e_p}| > \max(\alpha\sqrt{n}, 16)$  then
32    | Mark  $e_p$  as dense element;
33    | Move dense variables to end of ordering;
34  end
35 end
36 Post-processing::;
37 Apply elimination tree post-ordering;
38 Compute final permutation statistics;
39 return  $P$  and Info;
```

3. Implementation and Optimizations

Algorithm 5: Multilevel Graph Coarsening

```

input :Graph  $G = (V, E)$ 
output:Hierarchy of progressively coarser graphs

1 Hierarchy =  $[G]$  // Start with original graph
2 CurrentGraph =  $G$ ;
3 while  $|V(CurrentGraph)| > COARSENING_THRESHOLD$  do
4   Find heavy edge matching to preserve graph structure;
5   Matching = HeavyEdgeMatching(CurrentGraph);
6   Contract matched edges to create coarser graph;
7   CoarserGraph = ContractEdges(CurrentGraph, Matching);
8   Add to hierarchy;
9   Hierarchy.append(CoarserGraph);
10  CurrentGraph = CoarserGraph;
11 end
12 return Hierarchy;

```

Algorithm 6: Heavy Edge Matching Algorithm

```

input :Graph  $G$  with edge weights
output:Maximal matching favoring heavy edges

1 Matching = {};
2 Matched = {} // Track matched vertices
3 Sort edges by weight, centralness and degree for better matching quality;
4 SortedEdges = SortByWeight( $E(G)$ , descending = true);
5 foreach edge  $(u, v)$  in SortedEdges do
6   if  $u \notin \text{Matched}$  and  $v \notin \text{Matched}$  then
7     Matching.add( $(u, v)$ );
8     Matched.add( $u$ );
9     Matched.add( $v$ );
10    end
11 end
12 return Matching;

```

3.3. New Coarsening Approaches in Nested Dissection

This section presents a new coarsening algorithm developed in this thesis that improves upon the standard METIS approach.

The standard METIS coarsening approach uses heavy edge matching (SHEM) to create progressively smaller graphs while preserving structural properties. This work introduces a separator-aware heavy edge matching (SAHEM) algorithm that modifies the coarsening phase to better preserve separator structure during the multilevel nested

3. Implementation and Optimizations

dissection process.

The SAHEM algorithm extends the standard heavy edge matching by incorporating separator likelihood scores for each vertex. These scores are computed using two local structural features: normalized degree centrality and local clustering coefficient. High-degree vertices with low clustering coefficients are more likely to be separators in the graph, as they connect different regions without being part of densely connected communities.

For each vertex, the algorithm computes a separator score as:

$$\text{sep_score}(v) = \frac{\text{degree}(v)}{n} \times (1 - \text{clustering}(v))$$

The local clustering coefficient measures the fraction of possible edges that exist between a vertex's neighbors. Vertices with many neighbors but few connections among those neighbors typically serve as bridges between different graph regions.

During edge matching, SAHEM evaluates potential matches using an edge score that combines the edge weight with a penalty based on separator likelihood. For each candidate edge (u, v) , the algorithm computes:

$$\text{score}(u, v) = w(u, v) - \alpha \times \text{penalty}(u, v)$$

The penalty term considers three factors: degree difference between the vertices, neighborhood overlap, and the separator scores of both vertices. The degree difference penalty prevents matching vertices with very different connectivity patterns. The overlap penalty measures how many common neighbors the vertices share relative to their total neighborhoods. The separator penalty directly uses the precomputed separator scores.

This approach aims to avoid matching vertices that are likely to be separators with vertices in the interior of partitions. By preserving separator vertices during coarsening, the algorithm maintains clearer boundaries between different regions of the graph, which can lead to better quality separators when the graph is partitioned at the coarsest level and refined during projection.

3. Implementation and Optimizations

Algorithm 7: Separator-Aware Heavy Edge Matching (SAHEM)

```

input :Graph  $G = (V, E)$  with edge weights
output:Matching  $M$  and coarser graph

1 Compute separator likelihood scores:;
2 foreach vertex  $v \in V$  do
3   degree_centrality[ $v$ ] =  $\text{degree}(v)/|V|$ ;
4   clustering[ $v$ ] = clustering coefficient of  $v$ ;
5   sep_score[ $v$ ] = degree_centrality[ $v$ ]  $\times$  ( $1 - \text{clustering}[v]$ );
6 end

7 Initialize matching:;
8  $M = \emptyset$ ;
9 matched = {};
10 Sort vertices by degree (low to high);

11 Compute separator-aware matching:;
12 foreach vertex  $u$  in sorted order do
13   if  $u \notin \text{matched}$  then
14     best_score =  $-\infty$ ;
15     best_match = null;
16     foreach neighbor  $v$  of  $u$  do
17       if  $v \notin \text{matched}$  and weight constraints satisfied then
18         edge_score =  $w(u, v)$ ;
19         penalty = ComputePenalty( $u, v, \text{sep\_score}$ );
20         total_score = edge_score  $- \alpha \times \text{penalty}$ ;
21         if total_score > best_score then
22           best_score = total_score;
23           best_match =  $v$ ;
24         end
25       end
26     end
27     if best_match  $\neq \text{null}$  then
28        $M = M \cup \{(u, \text{best\_match})\}$ ;
29       matched = matched  $\cup \{u, \text{best\_match}\}$ ;
30     end
31   end
32 end
33 Create coarser graph by contracting matched edges;
34 return Coarser graph;

```

3. Implementation and Optimizations

Algorithm 8: Edge Cut Score Penalty Computation

```

input :Vertices  $u, v$ , separator scores
output:Penalty score

1 Degree difference penalty::
2 deg_diff = |degree( $u$ ) – degree( $v$ )| / max(degree( $u$ ), degree( $v$ ));
3 Neighborhood overlap penalty::
4 common = |neighbors( $u$ )  $\cap$  neighbors( $v$ )|;
5 total = |neighbors( $u$ )  $\cup$  neighbors( $v$ )|;
6 overlap_penalty = 1 – common/total;
7 Separator penalty::
8 sep_penalty = (sep_score[ $u$ ] + sep_score[ $v$ ])/2;
9 Combine penalties::
10 penalty = 0.4  $\times$  deg_diff + 0.4  $\times$  overlap_penalty + 0.2  $\times$  sep_penalty;
11 return penalty;

```

3.4. Hypergraph Based Ordering

This section presents a hypergraph-based ordering implementation developed in this thesis. The approach is inspired by the work of [14] and [19], which describe converting a symmetric matrix into Doubly-Bordered Block-Diagonal Form (DB) using hypergraph partitioning. This implementation extends their approach with an efficient method to find edge-clique cover and applies further ordering within the diagonal blocks.

The algorithm begins by constructing a standard graph representation of the symmetric matrix, where vertices correspond to matrix rows/columns and edges represent nonzero entries. From this graph, the algorithm computes an edge-clique cover, which identifies a set of cliques that collectively cover all edges in the graph. Each clique represents a group of vertices that are mutually connected.

The clique-node hypergraph transformation inverts the roles of vertices and cliques. Each clique becomes a node in the hypergraph, and each original graph vertex becomes a hyperedge connecting all cliques that contain it. This transformation enables the use of hypergraph partitioning tools to minimize the number of vertices that span multiple partitions, which naturally produces a vertex separator.

The hypergraph is partitioned using KaHyPar (Karlsruhe Hypergraph Partitioner) into k parts with a small allowed imbalance. The partitioning objective minimizes the connectivity metric, which counts hyperedges that connect nodes from different partitions. Each such hyperedge corresponds to a vertex in the original graph that belongs to multiple cliques assigned to different partitions.

After partitioning the clique-node hypergraph, the algorithm constructs the vertex separator by examining each original graph vertex. A vertex is assigned to partition i if all cliques containing it are assigned to partition i in the hypergraph partition. If a vertex belongs to cliques from multiple partitions, it becomes part of the separator. This

3. Implementation and Optimizations

produces k internal regions and one separator region.

To reduce fill-in during factorization, the algorithm applies an ordering method to each diagonal block. The implementation supports AMD, METIS, or nested dissection through the CHOLMOD library. Each diagonal block is extracted as a submatrix, and the chosen ordering algorithm reorders the vertices within that block to minimize fill-in during Cholesky factorization.

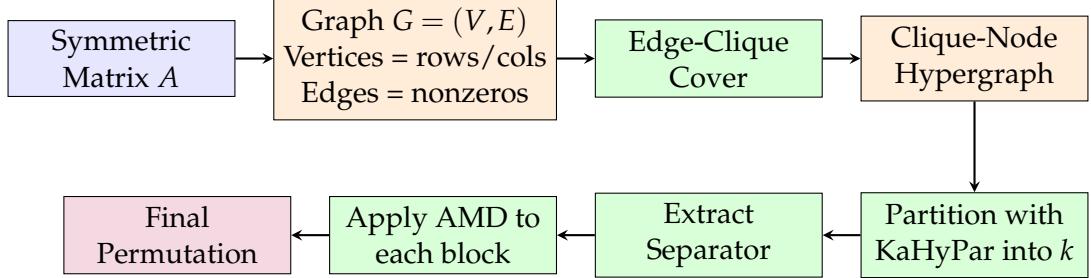


Figure 3.1.: Flowchart of the hypergraph-based symmetric DB form algorithm

The final permutation places all vertices from diagonal block 1, followed by diagonal block 2 through k , with separator vertices at the end. This produces a matrix in symmetric doubly-bordered block-diagonal form, where the diagonal blocks have been optimized for sparse factorization while the off-diagonal structure is confined to the border rows and columns corresponding to the separator.

3. Implementation and Optimizations

Algorithm 9: Hypergraph-Based Symmetric DB Form

input :Symmetric matrix A ($n \times n$), number of blocks k
output:Permutation P producing symmetric DB form

- 1 *Step 1: Create graph representation:;*
- 2 $G = (V, E)$ where $V = \{1, \dots, n\}$ and $(i, j) \in E$ if $A_{ij} \neq 0$;
- 3 *Step 2: Find edge-clique cover:;*
- 4 Cliques = FindEdgeCliqueCover(G);
- 5 Build mapping: vertex_to_cliques[v] = $\{c : v \in c, c \in \text{Cliques}\}$;
- 6 *Step 3: Build clique-node hypergraph:;*
 // Each clique becomes a hypergraph node
 // Each vertex becomes a hyperedge connecting its cliques
- 7 $H = (N, E_H)$ where $|N| = |\text{Cliques}|$;
- 8 **foreach** vertex $v \in V$ **do**
 9 | Create hyperedge e_v connecting nodes in vertex_to_cliques[v];
 10 | $E_H = E_H \cup \{e_v\}$;
- 11 **end**
- 12 *Step 4: Partition hypergraph:;*
- 13 clique_partition = PartitionHypergraph(H, k) using KaHyPar;
- 14 *Step 5: Extract vertex separator:;*
- 15 Initialize Parts[1.. k] = \emptyset , Separator = \emptyset ;
- 16 **foreach** vertex $v \in V$ **do**
 17 | partitions = {clique_partition[c] : $c \in \text{vertex_to_cliques}[v]$ };
 18 | **if** |partitions| = 1 **then**
 19 | | p = the single partition in partitions;
 20 | | $\text{Parts}[p] = \text{Parts}[p] \cup \{v\}$;
 21 | **end**
 22 | **else**
 23 | | Separator = Separator $\cup \{v\}$;
 24 | **end**
- 25 **end**
- 26 *Step 6: Apply AMD to each diagonal block:;*
- 27 **foreach** part $i = 1$ to k **do**
 28 | $A_i = A[\text{Parts}[i], \text{Parts}[i]]$ // Extract diagonal block
 29 | perm $_i$ = ApplyAMD(A_i) // Order using AMD/METIS/etc.
 30 | Reorder Parts[i] according to perm $_i$;
- 31 **end**
- 32 *Step 7: Build final permutation:;*
- 33 $P = [\text{Parts}[1], \text{Parts}[2], \dots, \text{Parts}[k], \text{Separator}]$;
- 34 **return** P ;

3. Implementation and Optimizations

Algorithm 10: Edge-Clique Cover (Greedy Heuristic)

```
input :Graph  $G = (V, E)$ 
output: Set of cliques covering all edges

1 Cliques =  $\emptyset$ ;
2 covered_edges =  $\emptyset$ ;
3 Find maximal cliques;;
4 maximal_cliques = all maximal cliques in  $G$ ;
5 Add cliques that cover new edges;;
6 foreach clique  $C$  in maximal_cliques do
7   clique_edges =  $\{(u, v) : u, v \in C, u < v, (u, v) \in E\}$ ;
8   new_edges = clique_edges \ covered_edges;
9   if new_edges  $\neq \emptyset$  then
10    | Cliques = Cliques  $\cup \{C\}$ ;
11    | covered_edges = covered_edges  $\cup$  clique_edges;
12  end
13 end
14 Cover remaining edges with 2-cliques;;
15 remaining =  $E \setminus$  covered_edges;
16 foreach edge  $(u, v) \in$  remaining do
17  | Cliques = Cliques  $\cup \{\{u, v\}\}$ ;
18 end
19 return Cliques;
```

3.5. GPU Implementation of RCM

Reverse Cuthill-McKee (RCM) can be thought of slightly differently and implemented given an efficient breadth-first search (BFS) function. The RCM algorithm is simply BFS with two modifications: (1) neighbors are visited in degree-sorted order, and (2) the final ordering is reversed. This structure allows us to wrap an existing GPU BFS implementation with preprocessing and postprocessing steps.

3. Implementation and Optimizations

Algorithm 11: GPU RCM Implementation

```

input :Graph  $G = (V, E)$  in CSR format
output:RCM permutation  $P$ 

1 Preprocessing: Sort adjacency lists;
2 foreach vertex  $u \in V$  in parallel on GPU do
3   | SortNeighborsByDegree( $u$ ) // Parallelly sort neighbors
4 end

5 Core traversal: Run BFS with sorted neighbors;
6  $order = []$ ;
7  $visited = \{\}$ ;
8 foreach component start vertex  $s$  (by increasing degree) do
9   | if  $s \notin visited$  then
10    |   | comp_order = GPU_BFS( $G, s$ ) // BFS from source  $s$ 
11    |   | visited = visited  $\cup$  comp_order;
12    |   | order = order + comp_order;
13   | end
14 end

15 Postprocessing: Reverse the ordering;
16  $P = \text{Reverse}(order)$ ;
17 return  $P$ ;

```

The GPU BFS was originally developed by NVIDIA in [20]. We work with one of the latest implementations of GPU BFS in [21]. The GPU BFS uses a level-synchronous approach that processes the graph one level at a time. Starting from a source vertex, the algorithm maintains a frontier of vertices to explore at the current level. Each iteration consists of two phases: expansion and contraction. In the expansion phase, all neighbors of frontier vertices are gathered in parallel by reading adjacency lists from CSR format. Since the adjacency lists were presorted by degree, neighbors are naturally visited in the correct RCM order. Multiple threads cooperatively read adjacency lists, with different parallelization strategies used based on vertex degree—single threads for low-degree vertices, entire warps (32 threads) for medium-degree vertices, and full thread blocks for high-degree vertices. In the contraction phase, the algorithm filters the gathered neighbors to remove duplicates and already-visited vertices using atomic compare-and-swap operations, producing the next level’s frontier. This process repeats until the frontier becomes empty, at which point the BFS traversal is complete.

This GPU RCM implementation adapts the parallel BFS from the NVIDIA CUDA SDK [21]. The modifications are just by having a preprocessing step before BFS and reversing the result after BFS.

Chapter 4

Results

In this chapter, we first describe our evaluation setup, which consists of the HPC cluster used for our experiments, the matrix datasets, and the algorithmic implementations evaluated. We then present a detailed analysis of various performance metrics, including execution time, memory usage, parallelization and matrix fill-in reduction.

4.1. Evaluation setup

Everything that is evaluated is ran on Fritz, a high-performance computing cluster at NHR@FAU.

4.1.1. Hardware Infrastructure

Fritz is a parallel CPU cluster operated by NHR@FAU, featuring Intel Ice Lake and Sapphire Rapids processors with an InfiniBand (IB) network and a Lustre-based parallel filesystem accessible under \$FASTTMP. The cluster configuration consists of:

| Nodes | CPUs and Cores | Memory | Slurm Partition |
|-------|---|--------|--------------------------|
| 992 | 2 × Intel Xeon Platinum 8360Y (Ice Lake) 2 × 36 cores @2.4 GHz | 256 GB | singlenode, multinode |
| 48 | 2 × Intel Xeon Platinum 8470 (Sapphire Rapids) 2 × 52 cores @2.0 GHz | 1 TB | spr1tb |
| 16 | 2 × Intel Xeon Platinum 8470 (Sapphire Rapids) 2 × 52 cores @2.0 GHz | 2 TB | spr2tb |

Table 4.1.: Fritz cluster node configuration

4. Results

The login nodes fritz[1-4] are equipped with $2 \times$ Intel Xeon Platinum 8360Y (Ice Lake) processors and 512 GB main memory. Additionally, the remote visualization node fviz1 features $2 \times$ Intel Xeon Platinum 8360Y processors with 1 TB main memory, one Nvidia A16 GPU, and 30 TB of local NVMe SSD storage.

4.2. Matrices dataset

We selected a diverse set of sparse matrices from the SuiteSparse Matrix Collection and our custom dataset of matrices encountered in Quantum Transport to evaluate our implementations. The SuiteSparse matrices are chosen to cover a variety of application domains, including computational fluid dynamics (CFD), chemistry, circuit simulation, graph analysis, optimization, and structural engineering. We have listed the matrices alongside their respective groups, number of vertices, and number of edges in Table 4.2. The Quantum Transport matrices are listed in Table 4.3.

| Graph name | Group | No. of vertices | No. of edges |
|------------------|---------------|-----------------|--------------|
| copter2 | CFD | 55476 | 407714 |
| ex10 | CFD | 2410 | 28625 |
| parabolic_fem | CFD | 525825 | 2100225 |
| 3Dspectralwave | Chemistry | 680943 | 17165766 |
| CO | Chemistry | 221119 | 3943588 |
| crystk03 | Chemistry | 24696 | 887937 |
| Ga19As19H42 | Chemistry | 133123 | 4508981 |
| Si87H76 | Chemistry | 240369 | 5451000 |
| SiH4 | Chemistry | 5041 | 88472 |
| circuit_3 | Circuit | 12127 | 48137 |
| G2_circuit | Circuit | 150102 | 438388 |
| memplus | Circuit | 17758 | 126150 |
| rajat06 | Circuit | 10922 | 28922 |
| rajat10 | Circuit | 30202 | 80202 |
| rajat15 | Circuit | 37261 | 443573 |
| 144 | Graphs | 144649 | 1074393 |
| 598a | Graphs | 110971 | 741934 |
| auto | Graphs | 448695 | 3314611 |
| ca-AstroPh | Graphs | 18772 | 198110 |
| citationCiteseer | Graphs | 268495 | 1156647 |
| delaunay_n20 | Graphs | 1048576 | 3145686 |
| dictionary28 | Graphs | 52652 | 89038 |
| boyd2 | Optimizations | 466316 | 890091 |
| c-55 | Optimizations | 32780 | 218115 |
| c-70 | Optimizations | 68924 | 363955 |
| gupta2 | Optimizations | 62064 | 2155175 |

Continued on next page

4. Results

| Graph name | Group | No. of vertices | No. of edges |
|------------|---------------|-----------------|--------------|
| gupta3 | Optimizations | 16783 | 4670105 |
| lpl1 | Optimizations | 32460 | 180248 |
| ncvxbqp1 | Optimizations | 50000 | 199984 |
| apache1 | Structural | 80800 | 311492 |
| apache2 | Structural | 715176 | 2766523 |
| nasasrb | Structural | 54870 | 1366097 |

Table 4.2.: SuiteSparse matrix dataset used for evaluation

Table 4.3.: Quantum Transport matrix dataset used for evaluation

| Graph name | No. of vertices | No. of edges | Description |
|-------------------------|-----------------|--------------|--|
| airpoll_conditional_st3 | 8499 | 1218651 | Spatio-temporal statistical Modelling |
| airpoll_prior_st3 | 8499 | 1205931 | Spatio-temporal statistical Modelling |
| cnt_cp2k | 9152 | 6154350 | DFT Hamiltonian of Carbon Nanotube |
| cnt_w90 | 768 | 71680 | DFT Hamiltonian of Carbon Nanotube |
| kmc_potential_0 | 403605 | 10007089 | Kinetic Monte Carlo for Device Modelling |
| kmc_potential_1 | 1632355 | 41208963 | Kinetic Monte Carlo for Device Modelling |
| qubit_fem_b4 | 116380 | 2517228 | FEM Matrix for Qubit |
| sinw_w90 | 7488 | 5310160 | DFT Hamiltonian of Silicon Nanowire |

4.3. Methodology

For our evaluation, we systematically assessed the performance of eleven different reordering algorithms across all matrices listed in Tables 4.2 and 4.3. The evaluated algorithms encompass both sequential and parallel approaches, ranging from classical degree-based methods to graph partitioning and hypergraph-based techniques:

1. Natural (Original matrix ordering - baseline)

2. **Classical Methods:**

- AMD (Approximate Minimum Degree)
- RCM (Reverse Cuthill-McKee)

3. **Graph Partitioning Methods:**

- METIS (Sequential graph partitioning-based reordering)
- **METIS+IC** (METIS with custom coarsening integration; in some places we refer to this as NDCUSTOM.)
- SCOTCH (Sequential graph partitioning)
- PT-SCOTCH (Parallel graph partitioning)
- ParMETIS (Parallel graph partitioning)

4. **Hypergraph-based Methods:**

4. Results

- HG-2 , HG-4 , HG-8 , HG-16 (Hypergraph-based reordering with block partition sizes of 2, 4, 8, and 16 vertices respectively. In some places we refer to these as DB-2, DB-4, DB-8, and DB-16 respectively.)

The hypergraph-based methods (HG-2, HG-4, HG-8, HG-16) employ different block partition sizes to explore the trade-off between computational complexity and reordering/parallelization quality. All these algorithms are described in detail in Chapter 3. The highlighted methods indicate our own implementations.

4.3.1. Performance Metrics

Our comprehensive evaluation measures five key performance indicators to assess the effectiveness of each reordering algorithm:

Fill-in Reduction: We measure the number of non-zero entries introduced during symbolic Cholesky factorization using the `cmpfillin` binary provided by the METIS library. This tool performs symbolic factorization (as described in Chapter 2) to count the additional non-zero entries without executing the actual numerical computation.

Reordering Time: The wall-clock time required to compute the reordering itself is measured using Python’s `time.perf_counter()` with microsecond precision. This overhead cost is crucial for understanding the practical applicability of each method.

Memory Usage: We implement a dedicated memory monitoring system using a separate thread that samples memory consumption every 50 milliseconds during reordering operations. The monitoring captures both peak memory usage (maximum RSS observed) and average memory consumption throughout the reordering process. Memory measurements are obtained via the `psutil` library, tracking the resident set size (RSS) of the process and computing the memory increase relative to a baseline measured before algorithm execution.

Parallelization Potential: We assess the elimination tree depth as an indicator of the inherent parallelism available in the factorization. As mentioned in Chapter 2, the elimination tree is constructed by identifying parent-child relationships in the symbolic factorization structure, where the tree depth represents the critical path length and thus the theoretical lower bound on parallel factorization time.

For memory profiling, we initially attempted to use the Massif profiler from Valgrind, but encountered issues with our Python-based implementations and the cluster environment. Consequently, we developed a custom memory measurement protocol that employs a thread-safe monitoring system to capture accurate memory usage patterns during reordering operations.

Our memory measurement implementation works by first establishing a baseline memory consumption before starting any reordering algorithm, recording the initial

4. Results

resident set size (RSS) using `psutil.Process().memory_info().rss`. During algorithm execution, a dedicated monitoring thread runs concurrently, sampling memory usage every 50 milliseconds throughout the entire operation. This thread continuously updates the maximum observed memory consumption and maintains a comprehensive list of all samples for subsequent average computation.

Upon completion of the reordering algorithm, we compute both the peak memory increase (calculated as the maximum observed RSS minus the baseline) and the average memory increase (computed as the mean of all samples minus the baseline), with results reported in kilobytes. For cases where the monitoring thread fails to capture meaningful data, such as very fast operations that complete before sufficient samples can be collected, we provide conservative estimates based on matrix characteristics, using approximately 100% of the matrix storage size as an upper bound for memory consumption.

All experiments are executed on the Fritz cluster to ensure consistent hardware conditions across all measurements. Each algorithm is tested on every matrix in our dataset, and results are averaged over five independent runs to mitigate variability.

4.4. Evaluation

For each matrix in our dataset with different reordering algorithms applied, we have collected all the aforementioned metrics, and these can be found in our appendix/supplementary material A. Here, we present results and visualizations for our underlying metrics for some of them.

Figure 4.1 presents a comparative analysis of **fill-in reduction** across different matrix categories, showing the ratio of fill-in to original non-zeros for representative matrices from each domain. This visualization demonstrates how reordering effectiveness varies significantly across different problem types and matrix structures.

4. Results

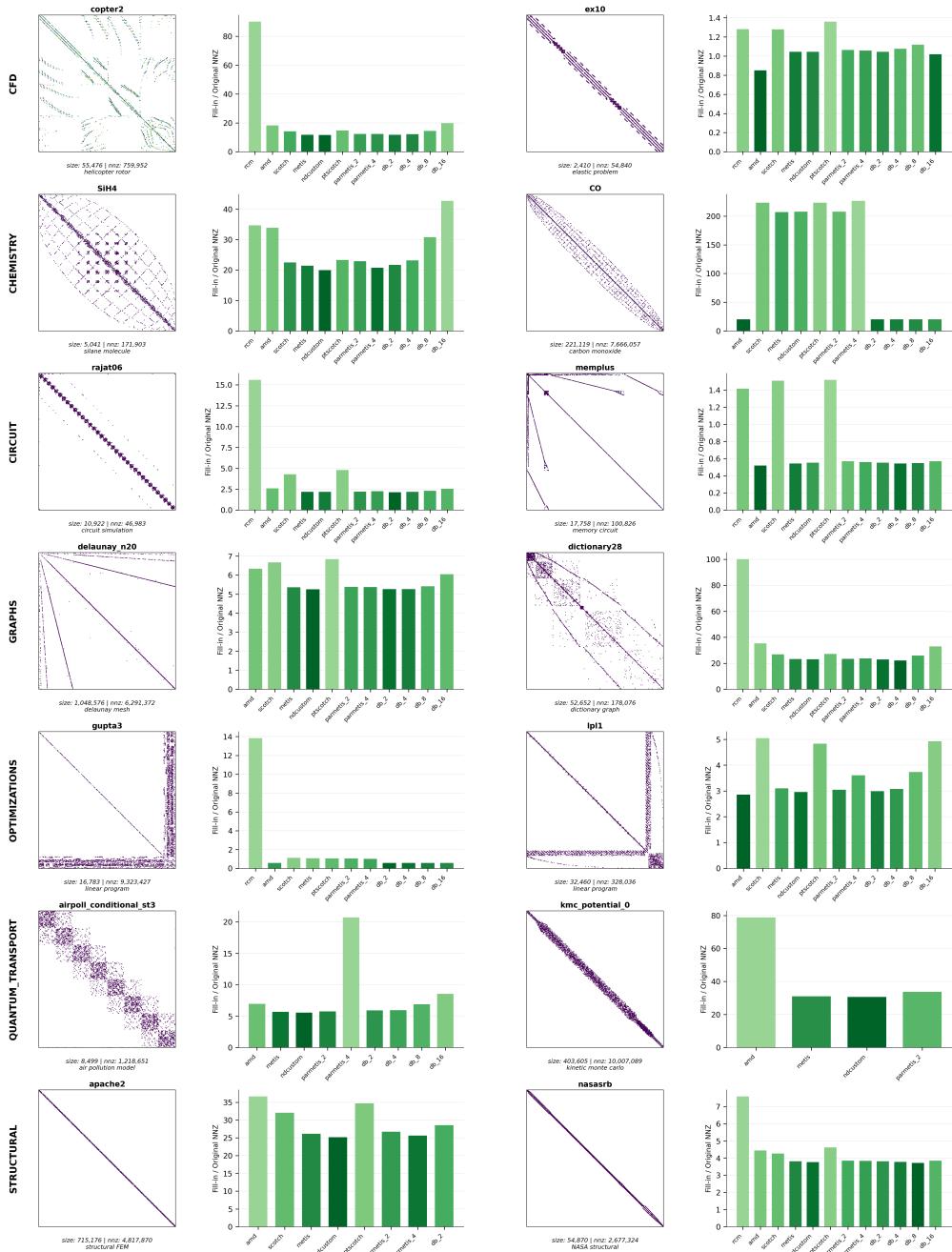


Figure 4.1.: Fill-in ratio (fill-in/original nnz) comparison across different matrix categories

Our experimental results reveal distinct performance patterns across different matrix domains and algorithm categories. The hypergraph-based methods, METIS, and our custom METIS implementation with improved coarsening (METIS+IC) show strong

4. Results

performance with lower fill-in reduction compared to classical methods. These methods achieve 30-60% lower fill-in values than traditional approaches like AMD and CO-LAMD across most domains. Among the hypergraph methods, smaller partition sizes (HG-2 and HG-4) typically produce optimal results, while larger partitions are ineffective. Our custom METIS+IC implementation demonstrates that enhanced coarsening strategies can achieve performance comparable to hypergraph methods. SCOTCH and PT-SCOTCH show moderate improvements over classical methods but do not match the quality of METIS-based approaches.

Domain-specific analysis reveals that hypergraph methods, METIS, and METIS+IC perform well in circuit and graph problems, where they achieve 35-50% fill-in reduction compared to AMD. For CFD matrices, these methods show modest but consistent improvements (10-30%), while chemistry matrices present scalability challenges where many classical methods fail on the largest instances, leaving primarily METIS-based or classical methods viable. The optimization and structural domains show variable performance depending on problem structure, with hypergraph methods and METIS variants providing 10-25% improvements in most cases.

Figure 4.2 presents the scaling behavior of reordering algorithms with respect to the number of non-zero entries. This graph shows the time required for each algorithm to complete and how it scales with increasing non-zero counts.

The **reordering time** results show notable performance differences across methods. RCM and AMD are the fastest options, with RCM typically completing in under one second (0.006-0.76s) across all matrix sizes and AMD performing well for small-to-medium matrices (0.046-2.09s). SCOTCH and METIS generally complete within reasonable timeframes (under 50 seconds), while parallel methods like ParMETIS and PT-SCOTCH have higher overhead that may be justified for larger problems. The hypergraph partitioning methods (HG variants) require considerably more time, with some executions taking over 270 seconds.

4. Results

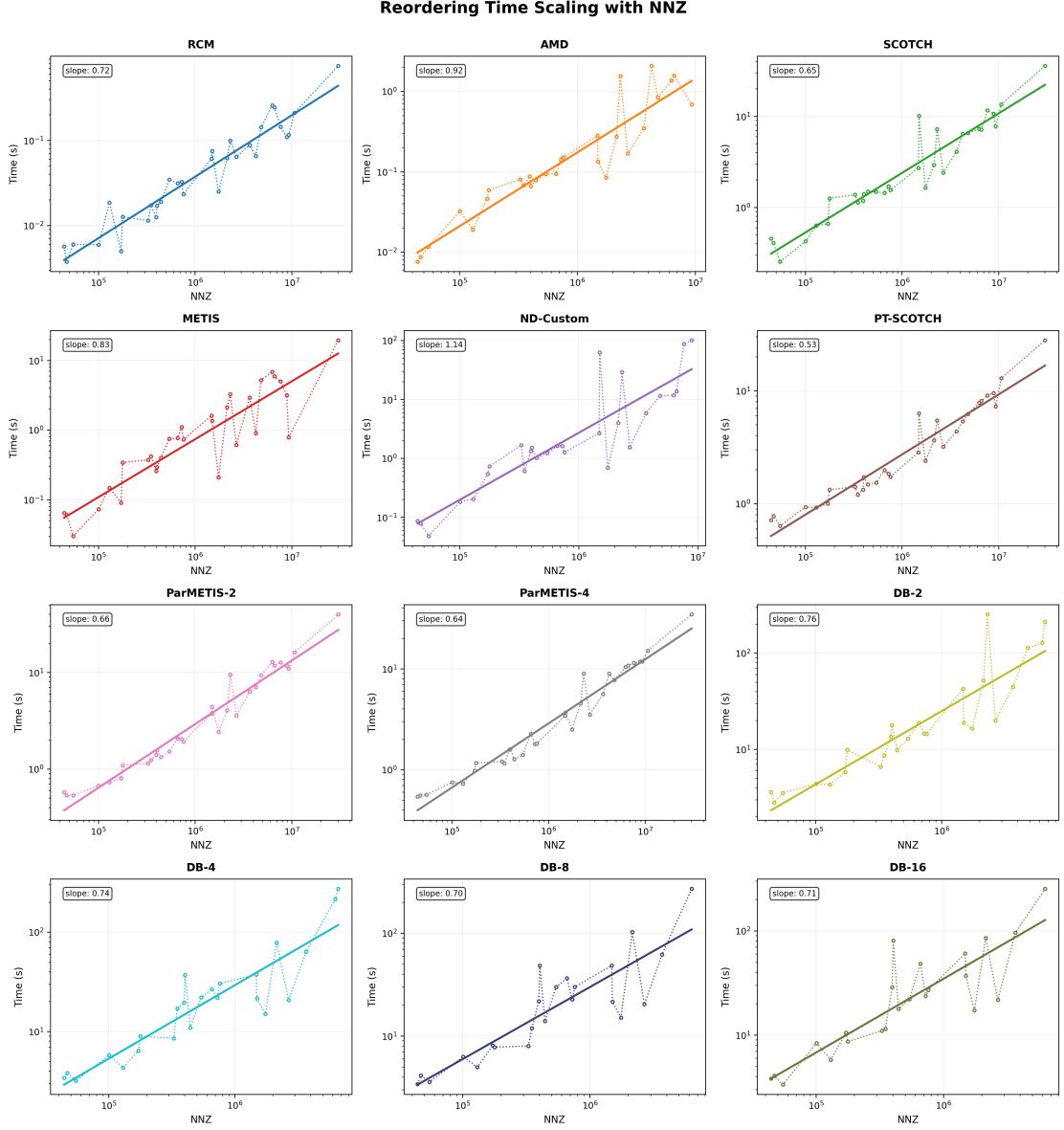


Figure 4.2.: Reordering time scaling with matrix size (number of non-zeros) for different algorithms. Each subplot shows the relationship between matrix size and computational time for various reordering methods.

Regarding scalability with increasing NNZ, the methods exhibit different growth patterns. AMD shows sublinear scaling (slope 0.52), which becomes more favorable as matrix size increases, while RCM maintains near-linear scaling (slope 0.72) with consistent performance characteristics. SCOTCH demonstrates reasonable scaling (slope 0.65), and the ParMETIS variants maintain moderate complexity (slopes 0.64-0.68) despite their higher base execution times. METIS+IC shows superlinear scaling (slope 1.14) due

4. Results

to its modified coarsening approach, which becomes problematic for very large matrices, while METIS approaches linear scaling (slope 0.81). The HG hypergraph methods show scaling slopes between 0.71-0.76, though their high absolute execution times remain a concern.

The results indicate that simpler methods like AMD and RCM provide favorable execution times compared to more complex approaches for large-scale sparse matrix reordering. While parallel methods may become advantageous for extremely large problems, the hypergraph partitioning approaches have higher computational costs relative to their fill-in benefits in typical sparse matrix applications.

Furthermore, we look at the **memory usage** of different reordering algorithms. Similar to reordering time, we plot memory scaling with respect to the number of non-zeros in Figure 4.3.

4. Results

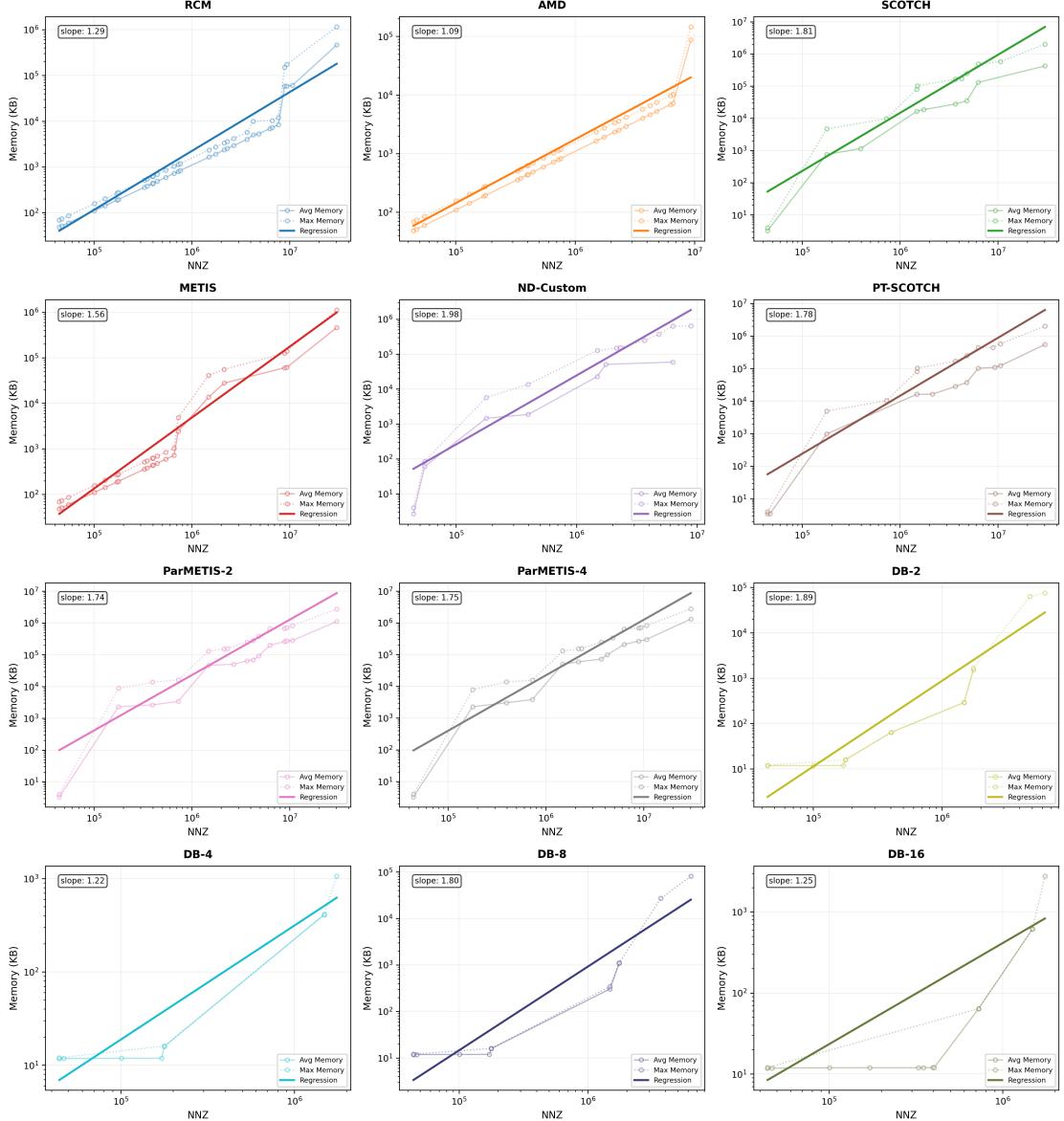


Figure 4.3.: Memory usage scaling with matrix size (number of non-zeros) for different algorithms. Each subplot shows the relationship between matrix size and memory consumption for various reordering methods.

The memory scaling analysis reveals distinct patterns across different reordering methods. Sequential methods like AMD and RCM demonstrate consistent scaling behavior with R^2 scores above 0.91. AMD exhibits near-linear memory scaling with a slope of 1.09, while RCM shows slightly super-linear scaling at 1.29.

The parallel and graph partitioning-based methods show steeper memory scaling. SCOTCH and PT-SCOTCH exhibit slopes around 1.8, while METIS demonstrates a

4. Results

slope of 1.56. The introduction of custom coarsening in METIS+IC increases the scaling exponent to 1.98 compared to standard METIS. ParMETIS variants (2 and 4 processes) show similar scaling behavior with slopes around 1.75, suggesting that the parallelization strategy introduces additional memory overhead that scales super-linearly with problem size.

The hypergraph-based HG methods present limited data, with only 4–7 valid monotonically increasing data points for maximum memory after filtering. The available data shows slopes ranging from 1.22 to 1.90, though the small sample sizes and lower R^2 scores (particularly HG-16 at 0.79) suggest these methods may exhibit less predictable memory behavior across the matrix suite tested. The filtered points indicate that these methods may not consistently scale memory usage with matrix size in the expected monotonic fashion.

We then now look at the elimination tree depth as a measure of parallelism. Similar to fill-in, we sample randomly two matrices from different domains to show the elimination tree depth for different reordering algorithms.

The **elimination tree depth** results provide insight into the parallelization potential of different reordering methods, as shallower trees enable more parallel execution during factorization. The parallel methods show advantages in reducing tree depth compared to sequential approaches. ParMETIS variants produce shallow elimination trees, with ParMETIS-2 and ParMETIS-4 achieving depths in the range of 14-42 across various matrices, while ParMETIS-8 and ParMETIS-16 maintain depths between 25-50. METIS also performs well for parallelization potential, producing trees with depths around 22-160, which represents a significant improvement over many sequential methods.

Sequential methods show mixed results for parallelization potential. AMD typically produces tree depths ranging from very shallow (4-20) for some matrices to quite deep (250-830) for others, making its parallelization benefits matrix-dependent. NDCUSTOM, being METIS with modified coarsening, generally produces similar tree depths to METIS, typically ranging from 24-171, though it can produce deeper trees (up to 839) for some matrices, suggesting that the coarsening modification does not improve parallelization potential across all cases. RCM produces the deepest trees among the methods tested, with depths exceeding 300-400 and reaching over 700 for some matrices, which limits parallel factorization opportunities. SCOTCH performs moderately, with tree depths generally ranging from 23-290, while PT-SCOTCH shows improved parallelization potential with depths typically between 54-328, though it can produce very deep trees exceeding 700.

The hypergraph partitioning methods (DB variants) show variable performance for parallelization, with tree depths ranging from 19-301 depending on the number of partitions used. DB-2 often produces deeper trees (49-603), while DB-16 tends toward moderate depths (66-830), though the relationship between partition count and tree depth is not always consistent. Natural ordering predictably produces very deep trees (often 200-800+), confirming the necessity of reordering for parallel factorization. The results indicate that while parallel reordering methods require more computational time, they provide substantial benefits for subsequent parallel factorization phases through significantly reduced elimination tree depths.

4. Results

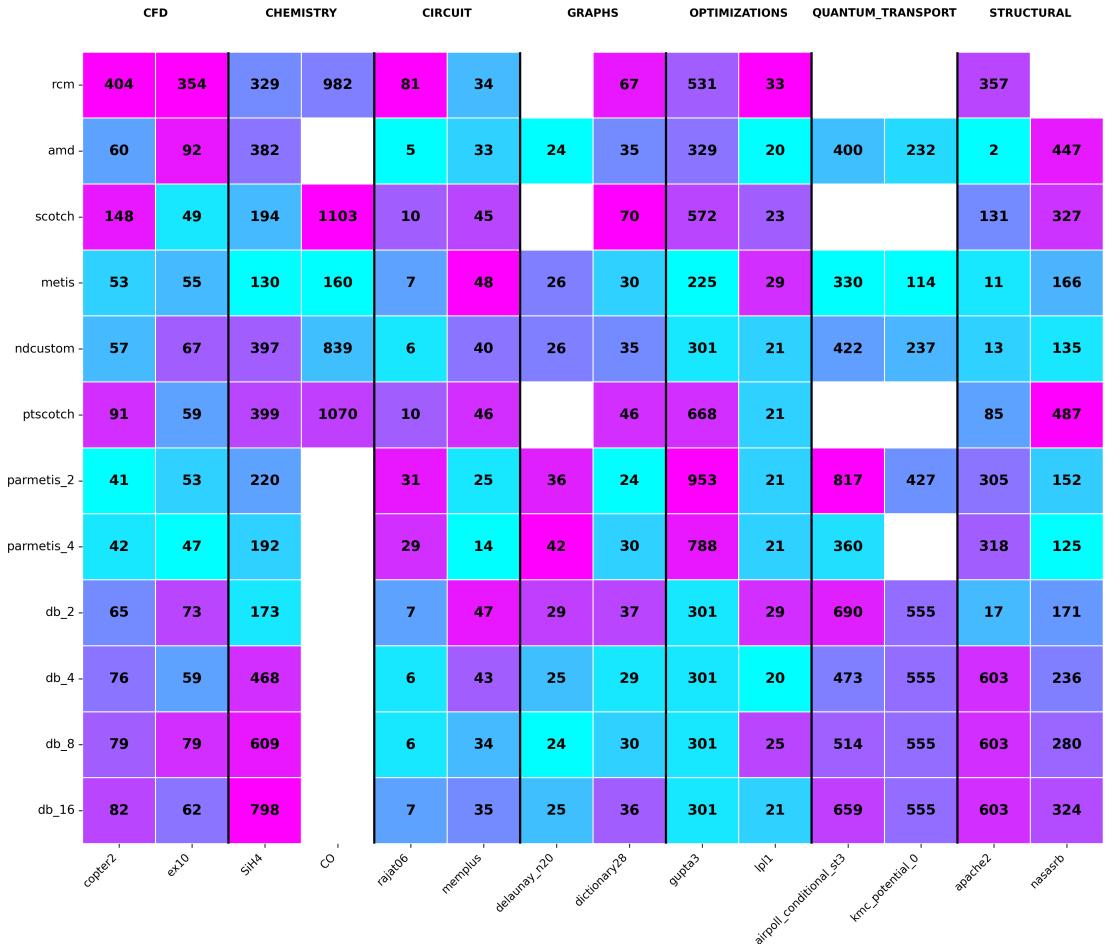


Figure 4.4.: Elimination tree depth comparison across different matrix categories for various reordering algorithms. Lower depth indicates better parallelization potential. Blue indicates lower depth (better), while pink indicates higher depth (worse).

4. Results

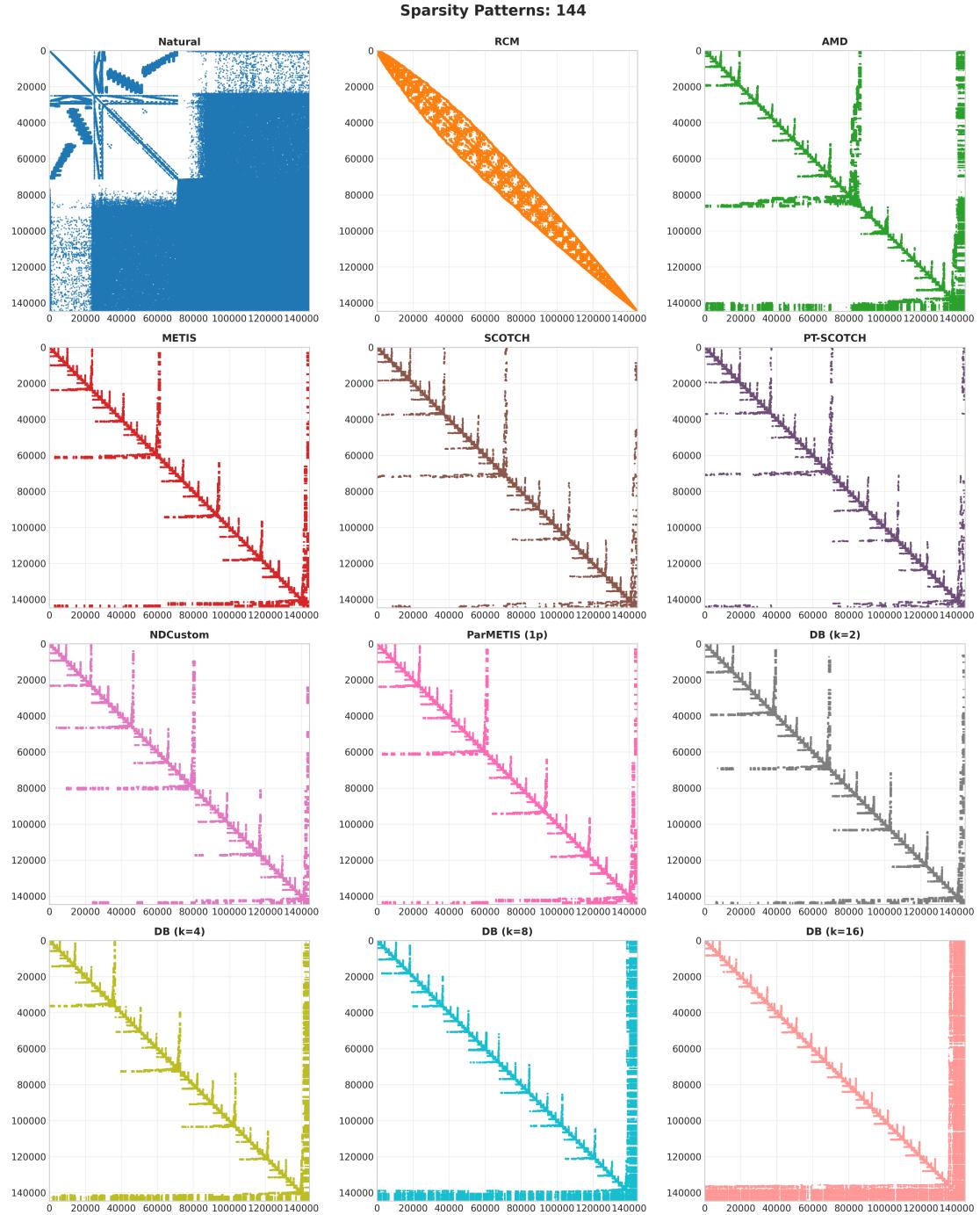


Figure 4.5.: Sparsity patterns of matrix 144 with different reordering algorithms applied. The natural ordering (top-left) shows a scattered pattern leading to high fill-in, while optimized reorderings (other panels) demonstrate more defined structures.

4. Results

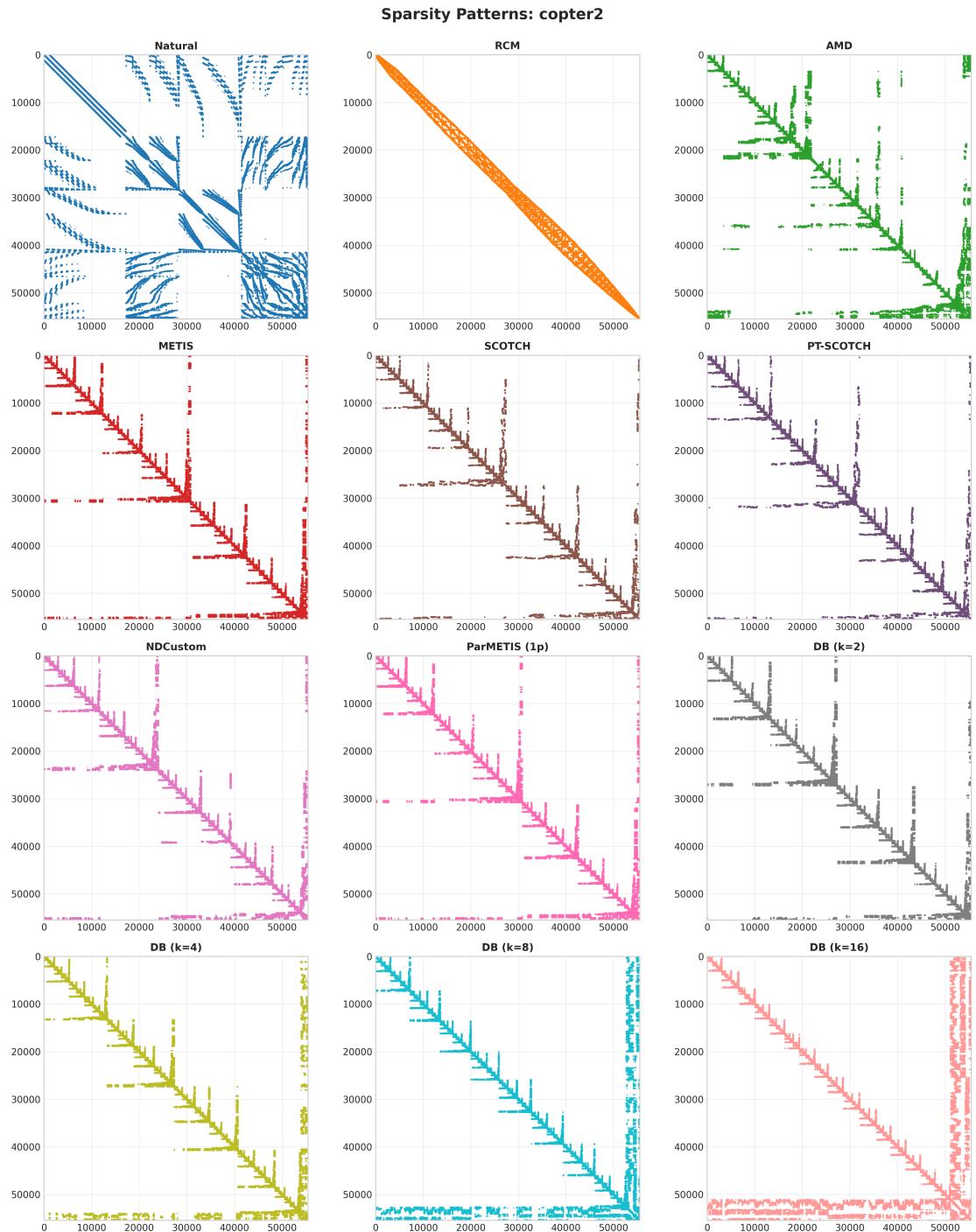


Figure 4.6.: Similarly, sparsity patterns of matrix copter2 with different reordering algorithms applied.

4. Results

4.4.1. GPU-accelerated RCM

In this section, we present the performance of our GPU-accelerated implementation of the Reverse Cuthill-McKee (RCM) algorithm compared to a traditional CPU-based implementation. We measure only the runtime of the RCM algorithm and not the fill-in, as the quality of the ordering produced by both implementations is identical. We test these implementations of a different hardware setup, as the Fritz cluster does not have suitable GPU nodes. Instead, we used the "alex" cluster, which provides GPU acceleration capabilities. The alex cluster consists of 44 nodes equipped with $2 \times$ AMD EPYC 7713 "Milan" processors running at 2.0 GHz, featuring $8 \times$ NVIDIA A100 GPUs (with 40 GB or 80 GB HBM2 memory), 1024 GB or 2048 GB system memory, 14TB local NVMe SSD storage, and HDR200 Infiniband networking. Additionally, 38 nodes contain the same processors with $8 \times$ NVIDIA A40 GPUs (48 GB DDR6 memory), 512 GB system memory, and 7 TB local NVMe SSD storage.

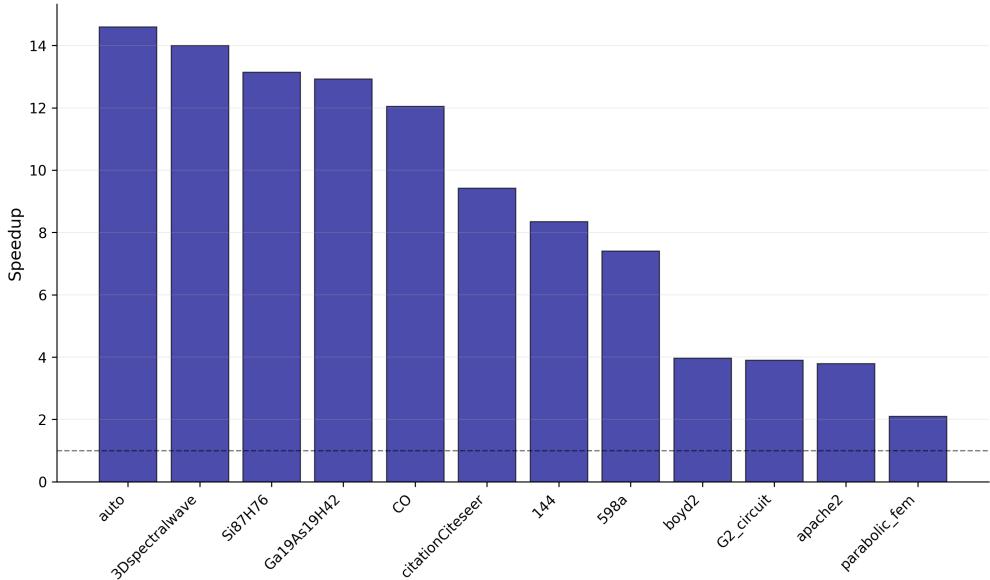


Figure 4.7.: Speedup of GPU-accelerated RCM compared to CPU-based RCM on large matrices (over 100K nodes).

For small matrices, the overhead of data transfer to and from the GPU negates any performance benefits, resulting in slower execution times compared to the CPU implementation. However, as matrix sizes increase, the GPU-accelerated RCM demonstrates significant performance improvements. In our experiments, we observe speedups ranging from 1.6x to 14.6x for matrices larger than 100,000 nodes. Some irregular matrices, such as c-70 and gupta3, exhibit particularly high speedups of 19.7x and 15.6x respectively, even though they are smaller than 100,000 nodes. Profiling indicated that the majority time of the GPU implementation for these matrices was spent in the pre-processing phases, suggesting that structures of the matrices can influence performance.

4.5. Current limitations

Our evaluation has important limitations that should be acknowledged. First, our analysis is not exhaustive—we evaluated only a subset of available sparse matrix reordering algorithms and libraries, focusing on the most commonly used and representative methods. This thesis does not cover a recent multi-threaded CPU implementation of AMD [22] or a multi-threaded Nested Dissection (with mt-METIS [23]), which could provide additional insights into performance trade-offs.

Additionally, our memory measurement system faced significant challenges: the monitoring thread frequently failed to capture meaningful data for fast-completing algorithms or when external binaries changed process IDs (particularly with hypergraph methods). For several large matrices, symbolic factorization failed due to memory constraints, preventing complete evaluation of all methods across our entire dataset.

The practical applicability of some methods is also limited by performance constraints. Hypergraph-based methods, while sometimes producing high-quality reorderings, proved computationally expensive and impractical for large matrices. Furthermore, our GPU acceleration efforts were limited to the RCM algorithm—we did not implement GPU versions of other reordering methods, which represents a significant opportunity for future performance improvements across the entire algorithm suite.

Chapter 5

Other Approaches

In this chapter, we present two methods for sparse matrix reordering that seemed to hold promise. These methods were found by myself to be either computationally difficult to scale for large matrices or were found to have some bottlenecks, but nevertheless represent interesting directions for future work.

5.1. Graph Reinforcement Learning for Reordering

As we know that sparse symmetric matrices can be represented as undirected graphs, it seems promising that this representation allows GNNs to naturally capture the local neighborhood relationships that classical ordering heuristics rely on, such as node degree and clustering patterns.

Traditional heuristics like minimum degree ordering make greedy decisions based on limited local information. There has been a few approaches for using machine learning to learn better heuristics, such as the work by [15], which uses CNN and they tackle large matrices by partitioning the graph.

GNNs, however, can propagate information across multiple hops in the graph, enabling each node to consider not just its immediate neighbors but also the broader structural context. It also allows us to have variable sized graphs. This multi-hop reasoning capability allows the network to anticipate how eliminating one node will affect distant parts of the matrix, potentially leading to more effective ordering decisions.

The message-passing architecture of GNNs naturally models the fill-in process during matrix factorization. When a node is eliminated, it creates new connections between its neighbors, which GNNs can represent through their aggregation and update mechanisms. The network can learn to predict these fill-in patterns and make elimination choices that minimize overall structural complexity.

The graph neural network, as shown in Figure 5.1, processes the sparse matrix represented as a graph, computing embeddings for each node based on features such as degree, clustering coefficient, and centrality measures. The actor network uses these embeddings to select which node to eliminate next, while the critic network estimates

5. Other Approaches

the value of the current state. The agent receives rewards based on the fill-in generated at each step, with penalties for creating new edges and bonuses for efficient elimination patterns.

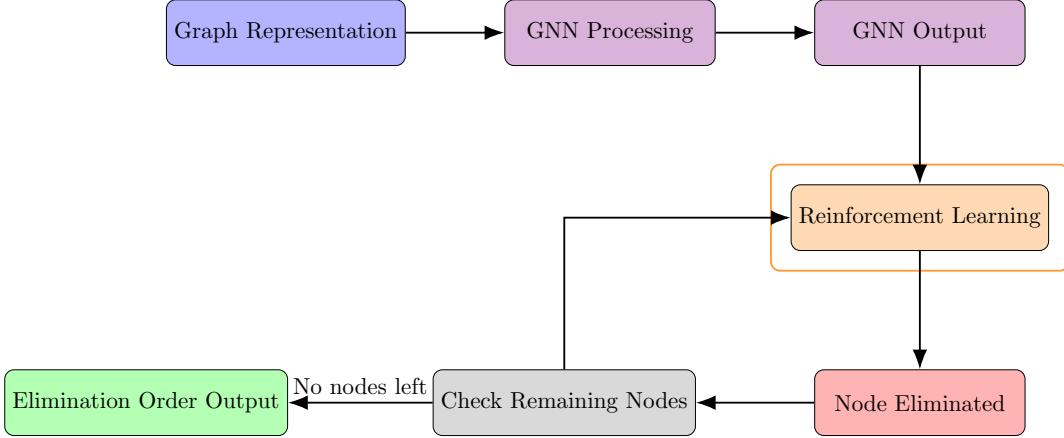


Figure 5.1.: Reinforcement Learning Framework for Sparse Matrix Reordering using GNNs

The model shows good and promising results on small matrices, achieving 9.8% improvement over RCM and 81.4% improvement over MinDegree on the ash85 matrix, and 28.4% and 51.0% improvements respectively on the bcsstk22 matrix. Furthermore, experiments on synthetic graphs with a few hundred nodes demonstrated good results, suggesting the approach can handle moderately sized problems. However, the method struggles to scale to very large matrices due to the computational complexity of training GNNs on large graphs. Future work can explore scaling techniques such as graph coarsening, hierarchical approaches, or coarser representations of the graph to make it feasible for larger matrices.

5.2. GPU Accelerated Nested Dissection

There is very limited work on GPU-accelerated reordering algorithms. I had presented earlier in this thesis, a GPU implementation of the RCM algorithm. Another promising approach is to implement the nested dissection algorithm on GPUs. One such approach is presented by [24], which leverages a GPU-based geometry processing library to perform fast graph partitioning, a key step in nested dissection. The method achieves significant speedups over the traditional CPU-based METIS library but the quality of the ordering is around 3 to 8 times worse than METIS, in their experiments.

The primary bottleneck in running nested dissection on GPUs is the recursive nature of the algorithm. Each level of recursion requires partitioning

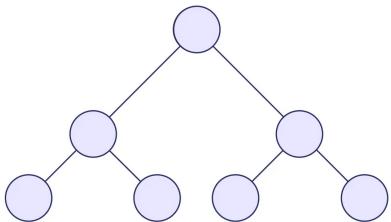


Figure 5.2.: Recursive calls

5. Other Approaches

the graph and identifying separators. We leveraged a graph processing library, Jet, which is a multilevel graph partitioning library designed for GPU [25]. The library comes with an efficient implementation of k-way partitioning done using a parallel multilevel approach. One can do a 2-way partitioning by setting $k=2$, but unsurprisingly, it performs worse in run-time as each call/partition

(circle in Figure 5.2) is done sequentially. Another approach was to use $k = 2^d$, where d is the depth of recursion such each subgraph reaches the threshold size after d levels of recursion. This approach is similar to the approach done using Hypergraph partitioning as mentioned in Chapter 3, in the way that the partition divides the graph into smaller subgraphs and the cumulative separators found are placed at the end, producing sparsity-pattern similar to hypergraph based methods in Chapter 3. This method, which is questionable to call it Nested Dissection, however produces orderings faster than METIS for large graphs but the quality of the ordering is still not comparable to METIS expectedly as you get better separators when you do recursive bisection than a single k-way partitioning. Such an approach produces orderings that are around 2 to 3 times worse than METIS in my experiments on large matrices (`parabolic_fem.graph` and `auto.graph`) but the speedups are significant around 5 to 6 times tested on the same environment as mentioned in Chapter 4.

Chapter 6

Conclusion and Future Work

This thesis presented a comprehensive evaluation of sparse symmetric matrix reordering algorithms across diverse matrix structures and application domains. Through systematic benchmarking on the Fritz HPC cluster, we assessed many reordering algorithms, ranging from classical degree-based methods to newer graph partitioning and hypergraph-based techniques, across multiple performance metrics including fill-in reduction, computational efficiency, memory consumption, and parallelization potential.

Our evaluation reveals that advanced methods such as METIS, SCOTCH, and hypergraph based approaches consistently outperform classical algorithms like AMD and RCM in terms of fill-in reduction, achieving 30-60% fewer fill-ins across most domains. Among the hypergraph methods, smaller partition sizes (HG-2 and HG-4) demonstrate superior effectiveness. Our custom METIS implementation with improved coarsening (METIS+IC) validates the benefits of enhanced coarsening strategies, producing results competitive with the best hypergraph methods. However, this improvement comes at a computational cost. Hypergraph methods require significantly more time (up to 270 seconds), while simpler methods like RCM complete in under one second.

In the course of this work, we implemented several efficient variants of existing algorithms. Our GPU-accelerated RCM implementation demonstrates substantial speedups over CPU-based versions for large matrices. The METIS+IC variant, incorporating refined coarsening strategies, achieves fill-in quality comparable to more complex hypergraph methods while maintaining reasonable execution times. We also explored hypergraph-based reordering methods that exploit block structures in matrices, showing promise particularly for circuit and graph problems.

Several promising directions remain for future investigation. Machine learning approaches, particularly graph neural networks (GNNs), show encouraging results on small matrices but require techniques like coarsening or hierarchical representations to scale to larger problems. Reordering algorithms for emerging architectures also warrant further research, while we implemented GPU-accelerated RCM, extending GPU implementations to nested dissection, minimum degree variants, and hypergraph methods could yield significant performance improvements. Finally, our evaluation re-

6. Conclusion and Future Work

veals that algorithm effectiveness varies significantly across matrix domains, suggesting that adaptive schemes which automatically select or tune algorithms based on matrix characteristics could improve performance across diverse applications.

Appendix **A**

Experimental Results

This appendix presents the complete experimental results for all matrix reordering algorithms evaluated in this thesis.

Table A.1 presents the number of nonzero entries after factorization for each reordering method. Table A.2 presents the computational time in seconds required to compute each reordering. Lower values indicate faster reordering algorithms. Table A.3 presents the peak memory usage in kilobytes during the reordering computation. Lower values indicate better memory efficiency. Table A.4 presents the average memory usage in kilobytes during the reordering computation. Table A.5 presents the depth of the elimination tree produced by each reordering. Lower depths generally indicate better potential for parallel factorization.

Empty or zero cells indicate that the algorithm either failed to complete, exceeded memory limits, timed out or some computation error. All measurements were performed on identical hardware configurations to ensure fair comparison. There are some meaningless values for memory usage, especially for hypergraph methods, for example showing as 4KB, very likely caused by failing the capture the memory usage for the corresponding PID process.

A. Experimental Results

| Matrix | Natural | RCM | AMD | METIS | NDCustom | ParMETIS-2 | ParMETIS-4 | ParMETIS-8 | SCOTCH | PT-SCOTCH | db-2 | db-4 | db-8 | db-16 |
|-------------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 144 | 1.8×10^9 | 9.3×10^8 | 4.7×10^7 | 4.6×10^7 | 4.7×10^7 | 4.8×10^7 | 5.1×10^7 | 5.3×10^7 | 4.7×10^7 | 4.7×10^7 | 5.2×10^7 | 6.8×10^7 | 6.8×10^7 | 6.8×10^7 |
| 3Dspectralwave | 1.8×10^9 | — | 1.3×10^9 | 1.3×10^9 | 1.4×10^9 | 1.4×10^9 | 1.5×10^9 | 1.5×10^9 | 2.0×10^9 | 1.8×10^9 |
| 598a | 1.6×10^8 | 4.6×10^7 | 4.6×10^7 | 2.5×10^7 | 2.5×10^7 | 2.6×10^7 | 2.6×10^7 | 2.9×10^7 | 2.6×10^7 | 2.6×10^7 | 2.6×10^7 | 2.9×10^7 | 2.9×10^7 | 4.1×10^7 |
| CO | 1.6×10^9 | 1.8×10^9 | 1.6×10^8 | 1.6×10^9 | 1.6×10^9 | 1.6×10^9 | 1.7×10^9 | 1.7×10^9 | 1.7×10^9 | 1.6×10^8 |
| G2_circuit | 1.0×10^9 | 1.2×10^8 | 9.9×10^8 | 6.4×10^6 | 6.4×10^6 | 6.4×10^6 | 6.6×10^6 | 6.7×10^6 | 7.8×10^6 | 8.0×10^8 | 7.9×10^8 | 8.2×10^8 | 8.2×10^8 | 9.0×10^8 |
| Ga19_Asi19H42 | — | — | — | 7.5×10^8 | 7.3×10^8 | 7.7×10^8 | 1.9×10^9 | 1.9×10^9 | 2.0×10^9 | 2.0×10^9 | 2.0×10^9 | — | — | — |
| Si87H76 | 5.8×10^6 | 5.8×10^6 | 5.8×10^6 | 3.7×10^6 | 3.4×10^6 | 3.7×10^6 | 3.9×10^6 | 3.9×10^6 | 4.0×10^6 | 3.7×10^6 | 4.0×10^6 | 5.3×10^6 | 7.4×10^6 | 7.4×10^6 |
| SiH4 | 6.8×10^6 | 4.4×10^7 | 1.3×10^7 | 1.0×10^7 | 9.9×10^6 | 1.0×10^7 | 1.0×10^7 | 1.3×10^7 | 1.3×10^7 | 1.0×10^7 | 1.0×10^7 | 9.5×10^6 | 1.1×10^7 | 1.1×10^7 |
| apache1 | — | — | 1.3×10^8 | 1.2×10^8 | 1.2×10^8 | 1.3×10^8 | 1.3×10^8 | 1.4×10^8 | 1.4×10^8 | 1.5×10^8 | 1.5×10^8 | 1.5×10^8 | — | — |
| apache2 | — | — | 5.7×10^8 | 2.2×10^8 | 2.4×10^8 | 2.4×10^8 | 2.4×10^8 | 2.4×10^8 | 2.3×10^8 | 8.0×10^5 |
| auto | 2.7×10^8 | 2.8×10^8 | 8.1×10^5 | 8.0×10^5 | 8.9×10^5 | 8.9×10^5 | 8.9×10^5 | 8.9×10^5 | 1.6×10^6 | 1.8×10^9 | 8.0×10^5 | 8.0×10^5 | 8.0×10^5 | 8.0×10^5 |
| boyd2 | 4.6×10^6 | 1.1×10^8 | 3.5×10^6 | 3.2×10^6 | 3.1×10^6 | 3.1×10^6 | 3.1×10^6 | 3.1×10^6 | 3.0×10^6 | 7.6×10^6 |
| c-55 | — | — | 7.2×10^8 | 8.0×10^6 | 8.2×10^8 | 8.0×10^6 | 8.2×10^8 | 8.2×10^8 | 1.0×10^7 | 1.1×10^7 |
| c-70 | — | — | 1.3×10^6 | 1.2×10^6 |
| ca_AstroPh | — | — | 2.7×10^4 | 2.6×10^4 | 2.9×10^4 | 2.9×10^4 | 2.9×10^4 | 3.0×10^4 | 2.9×10^4 | 4.6×10^4 | 4.6×10^4 | 4.7×10^4 | 2.8×10^4 | 2.9×10^4 |
| circuit_3 | 2.7×10^4 | 1.3×10^6 | 2.6×10^4 | 6.7×10^8 | 7.2×10^8 |
| citationCleseer | — | 6.9×10^5 | 1.4×10^7 | 8.9×10^6 | 8.9×10^6 | 9.2×10^6 | 9.2×10^6 | 9.2×10^6 | 9.4×10^6 | 1.1×10^7 | 1.1×10^7 | 9.3×10^6 | 1.1×10^7 | 1.5×10^7 |
| copier2 | 7.0×10^8 | 1.4×10^7 | 6.0×10^6 | 5.8×10^6 | 6.2×10^6 | 7.4×10^6 | 7.6×10^6 | 9.2×10^6 | 9.2×10^6 | 9.6×10^6 |
| crystk03 | 5.4×10^6 | 1.3×10^8 | 5.4×10^6 | 1.0×10^7 | 8.6×10^6 | 1.0×10^7 |
| delanauy_n20 | — | — | 9.4×10^5 | 3.4×10^7 | 3.3×10^7 | 3.3×10^7 | 3.4×10^7 | 3.4×10^7 | 4.2×10^7 | 4.3×10^7 | 4.3×10^7 | 3.3×10^7 | 3.4×10^7 | 3.8×10^7 |
| dictionary28 | 7.9×10^7 | 1.8×10^7 | 6.3×10^6 | 4.1×10^6 | 4.1×10^6 | 4.1×10^6 | 4.3×10^6 | 4.3×10^6 | 4.3×10^6 | 4.8×10^6 | 4.8×10^6 | 4.9×10^6 | 4.9×10^6 | 5.9×10^6 |
| ex10 | 9.1×10^4 | 7.0×10^4 | 4.7×10^4 | 5.7×10^4 | 5.8×10^4 | 7.0×10^4 | 7.5×10^4 | 7.5×10^4 | 5.7×10^4 | 5.6×10^4 |
| gupta2 | 5.4×10^8 | 4.7×10^8 | 6.0×10^6 | 5.8×10^6 | 6.2×10^6 | 7.4×10^6 | 7.6×10^6 | 5.4×10^8 | 5.4×10^8 | 5.4×10^8 |
| gupta3 | 5.4×10^6 | 1.3×10^8 | 5.4×10^6 | 1.0×10^7 | 8.6×10^6 | 9.9×10^6 | 9.9×10^6 | 5.4×10^6 | 5.4×10^6 | 5.4×10^6 |
| lpl | — | — | 1.4×10^5 | 5.5×10^4 | 5.6×10^4 | 5.6×10^4 | 5.6×10^4 | 5.6×10^4 | 5.7×10^4 | 1.5×10^5 | 1.5×10^5 | 5.6×10^4 | 5.5×10^4 | 5.7×10^4 |
| memplus | — | — | 1.2×10^7 | 1.2×10^7 | 1.0×10^7 |
| nasasrb | 1.7×10^7 | 2.0×10^6 | 4.2×10^6 | 1.9×10^6 | 2.5×10^7 | 2.9×10^7 | 2.9×10^7 | 2.5×10^7 | 2.6×10^7 | 3.1×10^7 |
| nevxbqp1 | — | — | 2.2×10^8 | 3.6×10^7 | 1.2×10^5 | 1.0×10^5 | 1.0×10^5 | 1.0×10^5 | 2.0×10^5 | 2.3×10^5 | 2.3×10^5 | 1.0×10^5 | 1.1×10^5 | 1.2×10^5 |
| parabolic_fem | — | 2.0×10^6 | 7.3×10^5 | 3.3×10^6 | 4.0×10^5 | 3.4×10^5 | 3.4×10^5 | 3.4×10^5 | 6.3×10^5 | 6.6×10^5 | 6.6×10^5 | 3.4×10^5 | 3.5×10^5 | 3.6×10^5 |
| raja06 | 7.4×10^6 | 3.3×10^6 | 4.0×10^5 | 7.7×10^5 | 7.6×10^5 | 7.7×10^5 | 7.7×10^5 | 7.7×10^5 | 1.0×10^6 | 1.1×10^6 | 1.1×10^6 | 7.9×10^5 | 8.5×10^5 | 1.1×10^6 |
| raja10 | 8.1×10^6 | — | 2.5×10^7 | 6.8×10^6 | 6.8×10^6 | 7.2×10^6 | 7.2×10^6 | 8.4×10^6 | 7.0×10^6 | 6.8×10^6 | 6.8×10^6 | 7.3×10^6 | 6.9×10^6 | 1.0×10^7 |
| raja15 | 1.2×10^7 | 2.5×10^7 | 9.0×10^6 | 8.4×10^6 | 8.4×10^6 | 7.1×10^6 | 7.1×10^6 | 8.3×10^6 | 7.1×10^6 | 6.0×10^6 | 6.0×10^6 | 6.8×10^6 | 7.1×10^6 | 9.9×10^6 |
| airpoll_conditional_st3 | 1.2×10^7 | 3.5×10^4 | 4.3×10^4 | 5.2×10^4 | 7.9×10^8 | 1.1×10^8 | 1.2×10^8 | 1.1×10^8 | 2.7×10^6 |
| airpoll_prior_st3 | 6.0×10^6 | 5.9×10^6 | 6.0×10^6 | 6.0×10^6 | 5.3×10^4 | 5.2×10^4 | 5.2×10^4 | 5.2×10^4 | 5.0×10^4 | 3.5×10^4 | 3.5×10^4 | 5.3×10^4 | 4.6×10^4 | 3.5×10^4 |
| cnt_cp2k | 3.5×10^4 | 2.7×10^6 | 5.0×10^6 | 5.0×10^6 | 5.0×10^6 | 4.9×10^6 | 4.9×10^6 | 4.7×10^6 |
| cnt_w90 | 3.8×10^8 | 2.7×10^6 | 3.2×10^8 | 3.8×10^8 | 1.2×10^8 | 1.1×10^8 | 1.1×10^8 | 1.1×10^8 | 2.7×10^6 |
| kmc_potential_0 | 1.1×10^9 | 2.7×10^6 | 5.0×10^6 | 5.0×10^6 | 5.0×10^6 | 4.9×10^6 | 4.9×10^6 | 4.7×10^6 |
| qubit_fem_b4 | 3.8×10^8 | 2.7×10^6 | 3.2×10^8 | 3.8×10^8 | 1.2×10^8 | 1.1×10^8 | 1.1×10^8 | 1.1×10^8 | 2.7×10^6 |
| sinrw_w90 | 2.7×10^6 | 5.0×10^6 | 5.0×10^6 | 5.0×10^6 | 4.9×10^6 | 4.9×10^6 | 4.7×10^6 |

Table A.1.: Fill-in (number of nonzeros after factorization) for each matrix and reordering algorithm.

A. Experimental Results

| Matrix | RCM | AMD | METIS | NDCustom | ParMETIS-2 | ParMETIS-4 | ParMETIS-8 | SCOTCH | PT-SCOTCH | db-2 | db-4 | db-8 | db-16 |
|-------------------------|-------|-------|--------|----------|------------|------------|------------|--------|-----------|---------|---------|---------|---------|
| 144 | 0.061 | 0.271 | 2.112 | 3.964 | 4.053 | 4.548 | 3.372 | 2.923 | 3.635 | 52.075 | 78.292 | 102.635 | 85.395 |
| 3Dspectralwave | 0.746 | 0.0 | 19.562 | 0.0 | 39.755 | 34.463 | 30.827 | 35.659 | 27.972 | 0.0 | 0.0 | 0.0 | 0.0 |
| 598a | 0.061 | 0.282 | 1.614 | 2.668 | 3.754 | 3.420 | 3.308 | 2.712 | 2.854 | 42.707 | 37.654 | 48.547 | 60.457 |
| CO | 0.144 | 0.0 | 4.985 | 86.865 | 12.658 | 11.394 | 10.552 | 11.558 | 9.062 | 0.0 | 0.0 | 0.0 | 0.0 |
| G2_circuit | 0.033 | 0.141 | 1.106 | 1.607 | 2.044 | 1.795 | 1.761 | 1.703 | 1.841 | 14.571 | 21.817 | 22.723 | 23.600 |
| Gal9As19H42 | 0.109 | 0.0 | 3.158 | 101.233 | 11.712 | 11.782 | 10.504 | 10.647 | 9.527 | 0.0 | 0.0 | 0.0 | 0.0 |
| S187H76 | 0.208 | 0.0 | 0.0 | 0.0 | 16.080 | 15.123 | 13.743 | 13.481 | 12.879 | 0.0 | 0.0 | 0.0 | 0.0 |
| SiH4 | 0.005 | 0.046 | 0.090 | 0.544 | 0.801 | 0.978 | 1.059 | 0.664 | 1.002 | 5.807 | 6.400 | 8.067 | 10.537 |
| apache1 | 0.035 | 0.094 | 0.750 | 1.231 | 1.549 | 1.407 | 1.416 | 1.482 | 1.535 | 12.918 | 21.983 | 30.059 | 22.131 |
| apache2 | 0.142 | 0.840 | 5.206 | 11.462 | 9.348 | 7.725 | 6.123 | 6.535 | 6.231 | 113.364 | 0.0 | 0.0 | 0.0 |
| auto | 0.243 | 1.564 | 5.927 | 13.647 | 11.815 | 10.773 | 9.497 | 7.134 | 8.113 | 212.413 | 273.232 | 0.0 | 0.0 |
| boyd2 | 0.074 | 0.134 | 1.364 | 62.858 | 4.383 | 3.674 | 3.291 | 10.078 | 6.315 | 18.90 | 21.477 | 21.597 | 36.917 |
| c-55 | 0.017 | 0.066 | 0.295 | 1.529 | 1.562 | 1.592 | 1.849 | 1.399 | 1.720 | 17.803 | 36.937 | 48.612 | 80.473 |
| c-70 | 0.031 | 0.094 | 0.766 | 1.627 | 2.068 | 2.268 | 2.307 | 1.443 | 1.984 | 18.863 | 26.485 | 36.458 | 48.367 |
| ca-AstroPh | 0.013 | 0.087 | 0.257 | 1.320 | 1.396 | 1.587 | 2.018 | 1.178 | 1.331 | 13.562 | 19.613 | 21.744 | 28.675 |
| circuit_3 | 0.006 | 0.064 | 0.087 | 0.544 | 0.540 | 0.540 | 0.627 | 0.454 | 0.718 | 3.602 | 3.406 | 3.829 | 3.829 |
| citationCiteSeer | 0.099 | 1.542 | 3.264 | 29.040 | 9.516 | 8.943 | 7.717 | 7.221 | 5.452 | 252.821 | 0.0 | 0.0 | 0.0 |
| copter2 | 0.023 | 0.151 | 0.733 | 1.269 | 1.924 | 1.819 | 1.861 | 1.547 | 1.734 | 14.522 | 30.263 | 30.062 | 27.043 |
| crystk03 | 0.025 | 0.084 | 0.210 | 0.690 | 2.424 | 2.507 | 2.788 | 1.644 | 2.396 | 16.430 | 15.065 | 15.106 | 17.250 |
| delaunay_n20 | 0.257 | 1.370 | 6.845 | 11.794 | 12.832 | 10.401 | 9.045 | 7.237 | 7.808 | 127.681 | 214.854 | 270.887 | 252.652 |
| dictionary28 | 0.013 | 0.059 | 0.343 | 0.729 | 1.091 | 1.164 | 1.269 | 1.258 | 1.330 | 9.888 | 18.965 | 7.783 | 8.655 |
| ex10 | 0.006 | 0.012 | 0.030 | 0.048 | 0.533 | 0.566 | 0.584 | 0.254 | 0.639 | 3.520 | 3.189 | 3.583 | 3.347 |
| gupta2 | 0.065 | 2.075 | 0.900 | 0.0 | 7.038 | 8.940 | 7.081 | 6.420 | 5.357 | 0.0 | 0.0 | 0.0 | 0.0 |
| gupta3 | 0.116 | 0.688 | 0.786 | 0.0 | 10.923 | 11.779 | 11.283 | 7.724 | 7.261 | 0.0 | 0.0 | 0.0 | 0.0 |
| lpl1 | 0.011 | 0.080 | 0.373 | 1.668 | 1.149 | 1.206 | 1.481 | 1.377 | 1.413 | 6.590 | 8.509 | 8.016 | 11.014 |
| mempbus | 0.006 | 0.032 | 0.073 | 0.183 | 0.674 | 0.674 | 0.832 | 0.424 | 0.936 | 4.374 | 5.776 | 6.287 | 8.373 |
| nasasrb | 0.064 | 0.169 | 0.613 | 1.539 | 3.587 | 3.526 | 2.807 | 2.406 | 3.199 | 20.019 | 20.685 | 20.413 | 21.757 |
| ncvxbqp1 | 0.017 | 0.068 | 0.420 | 0.604 | 1.238 | 1.154 | 1.169 | 1.134 | 1.206 | 8.660 | 17.101 | 11.923 | 11.470 |
| parabolic_fem | 0.088 | 0.348 | 2.937 | 5.803 | 6.332 | 5.607 | 5.099 | 4.100 | 4.370 | 44.500 | 63.501 | 61.983 | 95.454 |
| rajat6 | 0.004 | 0.009 | 0.061 | 0.078 | 0.532 | 0.555 | 0.607 | 0.408 | 0.779 | 2.802 | 3.827 | 4.136 | 4.066 |
| rajat10 | 0.018 | 0.019 | 0.148 | 0.204 | 0.730 | 0.727 | 0.767 | 0.631 | 0.925 | 4.317 | 4.326 | 4.974 | 5.796 |
| rajat15 | 0.019 | 0.078 | 0.399 | 1.008 | 1.330 | 1.268 | 1.355 | 1.495 | 1.486 | 9.842 | 10.911 | 14.023 | 17.835 |
| airpoll_conditional_st3 | 0.025 | 0.085 | 0.318 | 12.282 | 2.324 | 2.515 | 2.203 | 2.252 | 2.203 | 53.487 | 43.843 | 45.646 | 49.021 |
| airpoll_prior_st3 | 0.017 | 0.084 | 0.150 | 1.830 | 2.327 | 2.544 | 2.229 | 1.309 | 2.229 | 37.342 | 35.457 | 35.548 | 36.613 |
| cnt_cp2k | 0.064 | 0.277 | 0.717 | — | 6.220 | 7.732 | 5.158 | 4.865 | 5.158 | 0.0 | 0.0 | 0.0 | 0.0 |
| cnt_w90 | 0.005 | 0.010 | 0.148 | 0.204 | 0.730 | 0.727 | 0.767 | 0.631 | 0.925 | 4.317 | 4.326 | 4.974 | 5.796 |
| kmc_potential_0 | 0.332 | 1.094 | 5.709 | 61.167 | 13.940 | 13.016 | 10.575 | 11.459 | 10.575 | 0.0 | 0.0 | 0.0 | 0.0 |
| kmc_potential_1 | 1.515 | 0.0 | 0.0 | 317.549 | 55.548 | 49.505 | 34.982 | 37.262 | 34.982 | 0.0 | 0.0 | 0.0 | 0.0 |
| qubit_fem_b4 | 0.068 | 0.453 | 2.193 | 22.243 | 4.611 | 4.305 | 4.278 | 4.672 | 4.278 | 237.962 | 72.351 | 0.0 | 0.0 |
| sinw_w90 | 0.097 | 0.235 | 0.269 | 5.145 | 4.872 | 5.167 | 3.668 | 2.223 | 3.668 | 0.0 | 0.0 | 0.0 | 0.0 |

Table A.2.: Reordering time (seconds) for each matrix and reordering algorithm.

A. Experimental Results

| Matrix | RCM | AMD | METIS | NDCustom | ParMETIS-2 | ParMETIS-4 | ParMETIS-8 | SCOTCH | PT-SCOTCH | db-2 | db-4 | db-8 | db-16 |
|-------------------------|--------|--------|--------|----------|------------|------------|------------|---------|-----------|--------|--------|-------|-------|
| 144 | 3357 | 3357 | 55668 | 149388 | 151504 | 151220 | 150680 | 82500 | 85588 | 12 | 12 | 12 | 12 |
| 3Dspectralwave | 1150k | 0 | 1121k | 0 | 2750k | 2750k | 2014k | 2011k | 0 | 0 | 0 | 0 | 0 |
| 598a | 900 | 536 | 41088 | 127660 | 130220 | 129460 | 128680 | 80972 | 81096 | 292 | 412 | 344 | 616 |
| CO | 11978 | 0 | 11978 | 608508 | 580512 | 520376 | 609568 | 342212 | 342204 | 0 | 0 | 0 | 0 |
| G2_circuit | 1135 | 1135 | 4864 | 13312 | 16072 | 15480 | 14840 | 9776 | 10332 | 48 | 12 | 12 | 64 |
| Gal9As19H42 | 151000 | 0 | 125920 | 643008 | 679272 | 678784 | 643800 | 445332 | 445008 | 0 | 0 | 0 | 0 |
| S187H76 | 159656 | 0 | 0 | 0 | 824620 | 822872 | 781728 | 579816 | 5811828 | 0 | 0 | 0 | 0 |
| SiH4 | 269 | 269 | 847 | 847 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| apache1 | 847 | 847 | 7528 | 7528 | 373028 | 375932 | 337520 | 374576 | 246656 | 249608 | 63816 | 0 | 0 |
| apache2 | 7528 | 7528 | 10358 | 10358 | 52376 | 576724 | 525164 | 524616 | 315020 | 313500 | 53720 | 12 | 0 |
| auto | 10358 | 10358 | 44116 | 44116 | 9824 | 101928 | 100894 | 100392 | 102548 | 103336 | 108 | 12 | 104 |
| boyd2 | 2344 | 2344 | 630 | 630 | 4312 | 4556 | 4668 | 4728 | 4 | 4 | 64 | 12 | 12 |
| c-55 | 630 | 630 | 1030 | 1030 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| c-70 | 1030 | 1030 | 619 | 619 | 13620 | 13672 | 13884 | 13404 | 4128 | 3384 | 12 | 12 | 12 |
| ca-AstroPh | 619 | 619 | 69 | 69 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| circuit3 | 69 | 69 | 3615 | 3615 | 35420 | 154172 | 156948 | 155652 | 155100 | 101168 | 101444 | 12 | 0 |
| citationCiteSeer | 1187 | 1187 | 1187 | 1187 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 0 | 0 |
| copter2 | 2736 | 2736 | 10080 | 10080 | 101108 | 101100 | 114396 | 52984 | 5298 | 1652 | 1064 | 1104 | 2764 |
| crystk03 | 9830 | 9830 | 1528 | 637464 | 665980 | 640752 | 639516 | 490916 | 442864 | 75320 | 136 | 81396 | 296 |
| delanunay_n20 | 278 | 278 | 278 | 278 | 5728 | 5740 | 7824 | 7040 | 4668 | 4968 | 16 | 16 | 12 |
| dictionary28 | 86 | 86 | 9992 | 6638 | 6638 | 0 | 280616 | 247160 | 313728 | 171140 | 137372 | 4 | 12 |
| ex10 | 173344 | 145608 | 139524 | 0 | 712232 | 712240 | 675556 | 417288 | 417288 | 0 | 0 | 0 | 0 |
| guptia2 | 547 | 547 | 547 | 547 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| guptia3 | 5742 | 5742 | 19036 | 247344 | 249628 | 249176 | 277156 | 164000 | 169548 | 716 | 300 | 27156 | 304 |
| lp1 | 513 | 513 | 513 | 513 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| memplus | 158 | 158 | 4183 | 4183 | 124060 | 123856 | 123856 | 123776 | 63288 | 63748 | 12 | 12 | 12 |
| nasastib | 693 | 693 | 693 | 693 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| ncvxbqp1 | 1904 | 1904 | 1904 | 1904 | 69580 | 69920 | 69924 | 31688 | 31820 | 0 | 0 | 0 | 0 |
| parabolic_fem | 1884 | 1884 | 1884 | 1884 | 63548 | 63796 | 63740 | 25948 | 26368 | 25948 | 12 | 12 | 12 |
| rajat06 | 30944 | 9616 | 9616 | 0 | 383284 | 432120 | 236364 | 188276 | 236364 | 0 | 0 | 0 | 0 |
| rajat10 | 204 | 204 | 204 | 204 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| rajat15 | 693 | 693 | 693 | 693 | 4 | 4 | 4 | 4 | 4 | 4 | 12 | 12 | 12 |
| airpoll_conditional_st3 | 144440 | 78184 | 109184 | 899528 | 901560 | 938900 | 627736 | 668216 | 668216 | 0 | 0 | 0 | 0 |
| airpoll_prior_st3 | 921224 | 0 | 0 | 3863608 | 3863608 | 3862000 | 2778032 | 2773948 | 2778032 | 0 | 0 | 0 | 0 |
| cnt_cp2k | 43616 | 43616 | 24760 | 193464 | 193972 | 233052 | 109816 | 109204 | 109816 | 35500 | 76532 | 0 | 0 |
| qubit_fem_b4 | 76128 | 46336 | 48448 | 417968 | 417920 | 430000 | 249640 | 249648 | 249648 | 0 | 0 | 0 | 0 |
| sinw_w90 | 112 | 112 | 2724 | 3240 | 2984 | 248 | 248 | 368 | 368 | 1768 | 1128 | 1588 | 1588 |

Table A.3.: Maximum memory usage (KB) during reordering for each matrix and algorithm.

A. Experimental Results

| Matrix | RCM | AMD | METIS | NDCustom | ParMETIS-2 | ParMETIS-4 | ParMETIS-8 | SCOTCH | PT-SCOTCH | db-2 | db-4 | db-8 | db-16 |
|-------------------------|--------|-------|--------|----------|------------|------------|------------|--------|-----------|------|------|------|-------|
| 144 | 2350 | 27834 | 24469 | 40703 | 58832 | 48525 | 12380 | 16450 | 12.0 | 12.0 | 12.0 | 12.0 | 12.0 |
| 3Dspectralwave | 461793 | 0 | 460086 | 0 | 1120933 | 1317370 | 425597 | 548462 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 598a | 472 | 211 | 13696 | 22727 | 46393 | 50232 | 49445 | 16556 | 16260 | 292 | 411 | 305 | 615 |
| CO | 8385 | 0 | 8385 | 48144 | 175532 | 155282 | 219679 | 50329 | 58432 | 0.0 | 0.0 | 0.0 | 0.0 |
| G2_circuit | 795 | 2432 | 1784 | 3398 | 3839 | 3803 | 335 | 335 | 48 | 12 | 12 | 12 | 64 |
| Ga19As19t42 | 58637 | 0 | 60509 | 19684 | 260046 | 265948 | 245076 | 87241 | 98637 | 0.0 | 0.0 | 0.0 | 0.0 |
| S187H76 | 60738 | 0 | 0 | 0 | 280158 | 300024 | 298703 | 121061 | 123001 | 0.0 | 0.0 | 0.0 | 0.0 |
| SiH4 | 188 | 188 | 188 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| apache1 | 593 | 593 | 593 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| apache2 | 5270 | 5270 | 5270 | 26732 | 91929 | 94912 | 125461 | 34939 | 37490 | 70 | 0.0 | 0.0 | 0.0 |
| auto | 7251 | 7251 | 54253 | 198083 | 189256 | 206590 | 67046 | 71388 | 31 | 12 | 0.0 | 0.0 | 0.0 |
| boyd2 | 1641 | 1641 | 2376 | 3267 | 24068 | 19810 | 23174 | 5428 | 6840 | 108 | 12 | 104 | 12 |
| c-55 | 441 | 441 | 441 | 254 | 680 | 631 | 537 | 3 | 3 | 64 | 12 | 12 | 12 |
| c-70 | 721 | 721 | 721 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| ca-AstroPh | 433 | 433 | 433 | 1865 | 2634 | 3039 | 2391 | 1144 | 401 | 12 | 12 | 12 | 12 |
| circuit_3 | 48 | 48 | 48 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| citationCiteSeer | 2530 | 2530 | 11807 | 9830 | 34397 | 34500 | 29824 | 7963 | 10522 | 12 | 0.0 | 0.0 | 0.0 |
| copter2 | 831 | 831 | 831 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| crystk03 | 1915 | 1915 | 1915 | 50792 | 42837 | 42193 | 51024 | 18628 | 11867 | 1520 | 15 | 1100 | 294 |
| delanunay_n20 | 6881 | 6881 | 382 | 59146 | 199813 | 207967 | 242155 | 130707 | 102073 | 444 | 136 | 202 | 296 |
| dictionary28 | 195 | 195 | 195 | 1460 | 2297 | 2268 | 2000 | 754 | 984 | 16 | 16 | 16 | 12 |
| ex10 | 60 | 60 | 60 | 60 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| guptia2 | 4996 | 4647 | 4647 | 0 | 68475 | 98880 | 102083 | 24630 | 22404 | 0.0 | 0.0 | 0.0 | 0.0 |
| guptia3 | 58665 | 86992 | 61466 | 0 | 272550 | 260201 | 231491 | 115501 | 109466 | 0.0 | 0.0 | 0.0 | 0.0 |
| hp11 | 359 | 359 | 359 | 4 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| memplus | 110 | 110 | 110 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| nasastb | 2928 | 2928 | 2928 | 38015 | 50166 | 51248 | 38556 | 16725 | 12230 | 12 | 12 | 12 | 12 |
| ncvxbqp1 | 383 | 383 | 383 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| parabolic_fem | 4019 | 4019 | 9518 | 31722 | 64371 | 72374 | 108678 | 27836 | 28283 | 587 | 300 | 34 | 304 |
| raja06 | 51 | 51 | 51 | 2 | 3 | 3 | 3 | 3 | 4 | 12 | 12 | 12 | 12 |
| rajat10 | 143 | 143 | 143 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| rajat15 | 485 | 485 | 485 | 3 | 3 | 3 | 3 | 3 | 3 | 12 | 12 | 12 | 12 |
| airpoll_conditional_st3 | 1333 | 1333 | 1333 | 6057 | 24380 | 22278 | 6247 | 6186 | 6247 | 368 | 252 | 643 | 799 |
| airpoll_prior_st3 | 1319 | 1319 | 1319 | 28778 | 22287 | 20688 | 4871 | 9500 | 4871 | 0 | 443 | 264 | 28 |
| cnt_cp2k | 15472 | 6731 | 6731 | 0 | 119047 | 145990 | 47787 | 38493 | 47787 | 0 | 0 | 0 | 0 |
| cnt_w90 | 78 | 78 | 78 | 810 | 441 | 487 | 231 | 1 | 231 | 361 | 195 | 44 | 167 |
| kmc_potential_0 | 43898 | 32282 | 63399 | 73563 | 321702 | 392351 | 160563 | 149829 | 160563 | 0 | 0 | 0 | 0 |
| kmc_potential_1 | 438089 | 0 | 0 | 279053 | 1605689 | 1848255 | 869721 | 790627 | 869721 | 0 | 0 | 0 | 0 |
| qubit_fem_b4 | 21808 | 2 | 12380 | 14494 | 66102 | 90042 | 19132 | 17100 | 19132 | 375 | 121 | 0 | 0 |
| sinvw_w90 | 38064 | 15447 | 16151 | 144526 | 174394 | 171231 | 65498 | 122032 | 65498 | 0 | 0 | 0 | 0 |

Table A.4: Average memory usage (KB) during reordering for each matrix and algorithm.

A. Experimental Results

| Matrix | RCM | AMD | METIS | NDCustom | ParMETIS-2 | ParMETIS-4 | ParMETIS-8 | SCOTCH | PT-SCOTCH | db-2 | db-4 | db-8 | db-16 |
|-------------------------|------|-----|-------|----------|------------|------------|------------|--------|-----------|------|------|------|-------|
| 144 | 380 | 56 | 47 | 86 | 33 | 30 | 31 | 265 | 215 | 49 | 79 | 66 | 85 |
| 3Dspectralwave | 569 | 830 | 279 | 830 | 244 | 254 | 258 | 966 | — | 830 | 830 | 830 | 830 |
| 598a | 398 | 46 | 57 | 62 | 30 | 26 | 27 | 182 | 188 | 52 | 52 | 64 | 113 |
| CO | 982 | — | 160 | 839 | — | — | — | 1103 | 1070 | — | — | — | — |
| G2_circuit | 212 | 11 | 12 | 101 | 100 | 99 | 103 | 93 | 54 | 11 | 33 | 100 | 100 |
| Ga19As19H42 | — | — | 639 | 560 | — | — | — | — | 1009 | — | — | — | — |
| S187H76 | 1012 | — | — | — | — | — | — | 1039 | 929 | — | — | — | — |
| SiH4 | 329 | 382 | 130 | 397 | 220 | 192 | 165 | 194 | 399 | 173 | 468 | 609 | 798 |
| apache1 | 206 | 4 | 12 | 12 | 106 | 120 | 110 | 22 | 35 | 19 | 25 | 13 | 18 |
| apache2 | 357 | 2 | 11 | 13 | 305 | 318 | 332 | 131 | 85 | 17 | 603 | 603 | 603 |
| auto | 622 | 58 | 89 | 171 | 41 | 39 | 37 | 351 | 329 | 82 | 102 | 204 | 204 |
| boyd2 | 5 | 4 | 3 | 2 | 6 | 10 | 9 | 3 | 5 | 3 | 3 | 3 | 3 |
| c-55 | 10 | 23 | 37 | 26 | 31 | 30 | 26 | 33 | 25 | 29 | 27 | 23 | 19 |
| c-70 | 10 | 19 | 42 | 36 | 22 | 32 | 25 | 23 | 27 | 33 | 29 | 29 | 26 |
| ca-AstroPh | 184 | 105 | 112 | 124 | 52 | 56 | 49 | 183 | 246 | 113 | 116 | 115 | 105 |
| circuit_3 | 19 | 27 | 12 | 14 | 15 | 12 | 13 | 17 | 17 | 12 | 13 | 12 | 11 |
| citationCiteseer | 76 | 40 | 47 | 52 | 30 | 36 | 30 | 78 | 133 | 48 | 32 | 32 | 32 |
| copt2 | 404 | 60 | 53 | 57 | 41 | 42 | 43 | 148 | 91 | 65 | 76 | 79 | 82 |
| crystk03 | 719 | 251 | 126 | 125 | 212 | 273 | 164 | 290 | 328 | 173 | 179 | 203 | 218 |
| delanunay_n20 | — | 24 | 26 | 36 | 42 | 37 | 37 | — | — | 29 | 25 | 24 | 25 |
| dictionary28 | 67 | 35 | 30 | 35 | 24 | 30 | 33 | 70 | 46 | 37 | 29 | 30 | 36 |
| ex10 | 354 | 92 | 55 | 67 | 53 | 47 | 61 | 49 | 59 | 73 | 59 | 79 | 62 |
| gupta2 | 83 | 39 | 59 | 26 | 87 | 81 | 99 | 55 | 54 | 26 | 26 | 26 | 26 |
| gupta3 | 531 | 329 | 225 | 301 | 953 | 788 | 470 | 572 | 668 | 301 | 301 | 301 | 301 |
| hp11 | 33 | 20 | 29 | 21 | 21 | 22 | 23 | 23 | 21 | 29 | 20 | 25 | 21 |
| memplus | 34 | 33 | 48 | 40 | 25 | 14 | 19 | 45 | 46 | 47 | 43 | 34 | 35 |
| nasasib | — | 447 | 166 | 135 | 152 | 125 | 169 | 327 | 487 | 171 | 236 | 280 | 324 |
| ncvxbqp1 | 118 | 18 | 22 | 16 | 14 | 14 | 15 | 95 | 95 | 23 | 21 | 21 | 27 |
| parabolic_fem | — | 31 | 22 | 24 | 17 | 16 | 17 | 964 | 743 | 23 | 23 | 26 | 25 |
| raja06 | 81 | 5 | 7 | 6 | 31 | 29 | 24 | 10 | 10 | 7 | 6 | 6 | 7 |
| raja10 | 135 | 5 | 6 | 6 | 27 | 35 | 33 | 10 | 10 | 7 | 6 | 8 | 8 |
| raja15 | 77 | 81 | 62 | 66 | 31 | 33 | 35 | 92 | 99 | 60 | 56 | 59 | 58 |
| airpoll_conditional_st3 | 360 | 400 | 330 | 422 | 243 | 291 | 817 | 1096 | 817 | 659 | 690 | 473 | 514 |
| airpoll_prior_st3 | 428 | 452 | 390 | 358 | 223 | 235 | 775 | 806 | 775 | 676 | 611 | 517 | 760 |
| cnt_cp2k | — | — | 452 | 390 | — | — | — | — | — | — | — | — | — |
| cnt_w90 | 767 | 735 | 95 | 111 | 190 | 223 | 767 | 223 | 95 | 95 | 95 | 95 | 95 |
| kmc_potential_0 | — | 232 | 114 | 237 | 202 | 201 | 427 | 451 | 427 | 555 | 555 | 555 | 555 |
| kmc_potential_1 | — | 571 | 287 | 383 | 332 | 810 | 706 | 810 | 571 | 571 | 571 | 571 | 571 |
| qubit_fem_b4 | 248 | 116 | 45 | 60 | 82 | 75 | 140 | 104 | 140 | 98 | 80 | 225 | 225 |
| sinw_w90 | — | — | 910 | 908 | — | — | 910 | 961 | 910 | — | — | — | — |

Table A.5.: Elimination tree depth for each matrix and reordering algorithm.

Appendix **B**

Task Description

List of Figures

| | | |
|------|--|----|
| 1.1. | Sparsity pattern of a matrix from structural engineering <code>bcsstk32</code> (left) and $L + L^T$ (right) where L is the Cholesky factor of <code>bcsstk32</code> | 2 |
| 2.1. | Illustration of elimination tree construction for the given sparse symmetric matrix. The sparsity is denoted by filled black circles and the fill-in induced is denoted by hollow red circles. | 6 |
| 2.2. | Separator identification process. The separator vertices divide the graph into approximately equal subgraphs | 10 |
| 3.1. | Flowchart of the hypergraph-based symmetric DB form algorithm . . . | 20 |
| 4.1. | Fill-in ratio (fill-in/original nnz) comparison across different matrix categories | 29 |
| 4.2. | Reordering time scaling with matrix size (number of non-zeros) for different algorithms. Each subplot shows the relationship between matrix size and computational time for various reordering methods. | 31 |
| 4.3. | Memory usage scaling with matrix size (number of non-zeros) for different algorithms. Each subplot shows the relationship between matrix size and memory consumption for various reordering methods. | 33 |
| 4.4. | Elimination tree depth comparison across different matrix categories for various reordering algorithms. Lower depth indicates better parallelization potential. Blue indicates lower depth (better), while pink indicates higher depth (worse). | 35 |
| 4.5. | Sparsity patterns of matrix 144 with different reordering algorithms applied. The natural ordering (top-left) shows a scattered pattern leading to high fill-in, while optimized reorderings (other panels) demonstrate more defined structures. | 36 |
| 4.6. | Similarly, sparsity patterns of matrix copter2 with different reordering algorithms applied. | 37 |
| 4.7. | Speedup of GPU-accelerated RCM compared to CPU-based RCM on large matrices (over 100K nodes). | 38 |

List of Figures

| | |
|---|----|
| 5.1. Reinforcement Learning Framework for Sparse Matrix Reordering using GNNs | 41 |
| 5.2. Recursive calls | 41 |

List of Tables

| | | |
|------|--|----|
| 4.1. | Fritz cluster node configuration | 24 |
| 4.2. | SuiteSparse matrix dataset used for evaluation | 26 |
| 4.3. | Quantum Transport matrix dataset used for evaluation | 26 |
| A.1. | Fill-in (number of nonzeros after factorization) for each matrix and reordering algorithm. | 46 |
| A.2. | Reordering time (seconds) for each matrix and reordering algorithm. | 47 |
| A.3. | Maximum memory usage (KB) during reordering for each matrix and algorithm. | 48 |
| A.4. | Average memory usage (KB) during reordering for each matrix and algorithm. | 49 |
| A.5. | Elimination tree depth for each matrix and reordering algorithm. | 50 |

List of Algorithms

| | | |
|-----|---|----|
| 1. | Direct Construction Method | 6 |
| 2. | Elimination Tree Symbolic Factorization | 7 |
| 3. | SuiteSparse RCM Algorithm | 13 |
| 4. | SuiteSparse AMD Algorithm | 15 |
| 5. | Multilevel Graph Coarsening | 16 |
| 6. | Heavy Edge Matching Algorithm | 16 |
| 7. | Separator-Aware Heavy Edge Matching (SAHEM) | 18 |
| 8. | Edge Cut Score Penalty Computation | 19 |
| 9. | Hypergraph-Based Symmetric DB Form | 21 |
| 10. | Edge-Clique Cover (Greedy Heuristic) | 22 |
| 11. | GPU RCM Implementation | 23 |

Bibliography

- [1] J. Scott and M. Tůma, "An Introduction to Sparse Matrices," in *Algorithms for Sparse Linear Systems*, J. Scott and M. Tůma, Eds. Cham: Springer International Publishing, 2023, pp. 1–5. [Online]. Available: https://doi.org/10.1007/978-3-031-25820-6_1
- [2] H. Schaeffer, R. Caflisch, C. D. Hauck, and S. Osher, "Sparse dynamics for partial differential equations," *Proceedings of the National Academy of Sciences*, vol. 110, no. 17, pp. 6634–6639, Apr. 2013, publisher: Proceedings of the National Academy of Sciences. [Online]. Available: <https://www.pnas.org/doi/10.1073/pnas.1302752110>
- [3] J. R. Gilbert and R. E. Tarjan, "The analysis of a nested dissection algorithm," *Numerische Mathematik*, vol. 50, no. 4, pp. 377–404, Jul. 1986. [Online]. Available: <https://doi.org/10.1007/BF01396660>
- [4] M. Yannakakis, "Computing the Minimum Fill-In is NP-Complete," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 1, pp. 77–79, Mar. 1981, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: <https://pubs.siam.org/doi/10.1137/0602010>
- [5] A. Lim, B. Rodrigues, and F. Xiao, "Heuristics for matrix bandwidth reduction," *European Journal of Operational Research*, vol. 174, no. 1, pp. 69–91, 2006.
- [6] C. H. Papadimitriou, "The NP-Completeness of the bandwidth minimization problem," *Computing*, vol. 16, no. 3, pp. 263–270, Sep. 1976. [Online]. Available: <https://doi.org/10.1007/BF02280884>
- [7] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172.
- [8] A. George and J. W. H. Liu, "Computer solution of large sparse positive definite systems," *Prentice-Hall Series in Computational Mathematics*, 1981.
- [9] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

Bibliography

- [10] J. W. H. Liu, "Modification of the minimum-degree algorithm by multiple elimination," *ACM Transactions on Mathematical Software*, vol. 11, no. 2, pp. 141–153, 1985.
- [11] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.
- [12] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized Nested Dissection," *SIAM Journal on Numerical Analysis*, vol. 16, no. 2, pp. 346–358, Apr. 1979, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: <https://pubs.siam.org/doi/10.1137/0716027>
- [13] A. George, "Nested Dissection of a Regular Finite Element Mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, Apr. 1973, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: <https://pubs.siam.org/doi/10.1137/0710032>
- [14] U. V. Çatalyürek, C. Aykanat, and E. Kayaaslan, "Hypergraph Partitioning-Based Fill-Reducing Ordering for Symmetric Matrices," *SIAM Journal on Scientific Computing*, vol. 33, no. 4, pp. 1996–2023, Jan. 2011, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: <https://pubs.siam.org/doi/10.1137/090757575>
- [15] A. Dasgupta and P. Kumar, "Alpha Elimination: Using Deep Reinforcement Learning to Reduce Fill-In During Sparse Matrix Decomposition," in *Machine Learning and Knowledge Discovery in Databases: Research Track: European Conference, ECML PKDD 2023, Turin, Italy, September 18–22, 2023, Proceedings, Part IV*. Berlin, Heidelberg: Springer-Verlag, Sep. 2023, pp. 472–488. [Online]. Available: https://doi.org/10.1007/978-3-031-43421-1_28
- [16] T. Davis, "DrTimothyAldenDavis/SuiteSparse," Oct. 2025, original-date: 2019-10-19T17:43:53Z. [Online]. Available: <https://github.com/DrTimothyAldenDavis/SuiteSparse>
- [17] "AldenMath – Mathematical software by Tim Davis." [Online]. Available: <https://aldenmath.com/>
- [18] P. R. Amestoy, T. A. Davis, and I. S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software*, vol. 30, no. 3, pp. 381–388, Sep. 2004.
- [19] O. Selvitopi, S. Acer, M. Manguoğlu, and C. Aykanat, "The Effect of Various Sparsity Structures on Parallelism and Algorithms to Reveal Those Structures," in *Parallel Algorithms in Computational Science and Engineering*, A. Grama and A. H. Sameh, Eds. Cham: Springer International Publishing, 2020, pp. 35–62. [Online]. Available: https://doi.org/10.1007/978-3-030-43736-7_2

Bibliography

- [20] D. Merrill, "Scalable GPU Graph Traversal."
- [21] P. Kaleta, "kaletap/bfs-cuda-gpu," May 2025, original-date: 2019-11-14T12:55:50Z. [Online]. Available: <https://github.com/kaletap/bfs-cuda-gpu>
- [22] Y.-H. Chang, A. Buluç, and J. Demmel, "Parallelizing the approximate minimum degree ordering algorithm: Strategies and evaluation," 2025. [Online]. Available: <https://arxiv.org/abs/2504.17097>
- [23] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 225–236.
- [24] C. Yuan, "Fast Sparse Matrix Reordering on GPU for Cholesky Based Solvers."
- [25] M. S. Gilbert, K. Madduri, E. G. Boman, and S. Rajamanickam, "Jet: Multilevel Graph Partitioning on Graphics Processing Units," *SIAM Journal on Scientific Computing*, vol. 46, no. 5, pp. B700–B724, Oct. 2024, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: <https://pubs.siam.org/doi/10.1137/23M1559129>
- [26] H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, Apr. 2008.
- [27] D. Hankerson, S. Vanstone, and A. Menezes, *Guide to Elliptic Curve Cryptography*, ser. Springer Professional Computing. Springer, 2004.
- [28] NIST, *Advanced Encryption Standard (AES) (FIPS PUB 197)*, National Institute of Standards and Technology, Nov. 2001.
- [29] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, "OCB: A block-cipher mode of operation for efficient authenticated encryption," in *ACM Conference on Computer and Communications Security*, 2001, pp. 196–205.
- [30] Xilinx. (2011, Nov.) Virtex-6 FPGA Configuration User Guide. UG360 (v3.4). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug360.pdf
- [31] ——. (2011, Oct.) 7 Series FPGAs Configuration User Guide. UG470 (v1.2). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf
- [32] Wikipedia, "Isaac Newton," accessed October 1, 2012. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Isaac_Newton&oldid=514997436