Graph Partitioning, Ordering, and Clustering for Multicore Architectures

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Dominique LaSalle

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Dr. George Karypis, Advisor

December, 2015

Acknowledgements

This thesis is the result of a significant investment of time and effort, and would not have been possible without the help and support of many people.

First and foremost, I would first like to thank my advisor, George, for teaching me the value of hard work, high standards, and coffee. He has spent countless hours helping me grow as a researcher and engineer. I am deeply endebted to him, and consider myself very lucky to have had such a talented advisor.

I would like to thank Professors Victoria Interrante, Yousef Saad, John Sartori, Shashi Shekhar, and Antonia Zhai for taking the time to serve on my preliminary exam and thesis committees.

I am thankful to have two amazing brothers, Andre and Deon. The support and many needed diversions they have provided me with during graduate school have been invaluable. They have gotten me out of trouble, and have gotten me into it. I could not ask for two better men to call my brothers.

I owe a great deal of thanks to the past and present members of Karypis Lab: Asmaa, Agi, Chris, David, Eva, Fan, Haoji, Jeremy, Kevin, Mohit, Rezwan, Santosh, Sara, Saurav, Shaden, and Xia. They have helped me grow as a person and survive the many trials and tribulations of graduate school. I am honored to have spent my time with such fine men and women, and I will miss them all dearly.

I would like to thank the wonderful staff at the Department of Computer Science, the Digital Technology Center, and the Minnesota Supercomputing Institute at the University of Minnesota for providing assistance, facilities, and other resources for my research.

Lastly, I would like to thank the great state of Minnesota, which has given me an everlasting love of the outdoors and all of the challenges that life brings.

Dedication

To my parents, who have never waivered in giving me all of their love and support, despite the many years of household damages I may have caused.

Abstract

The terms dual-core and quad-core have become ubiquitous for modern computer processors. Where processors previously had a single powerful processing core, they now have multiple lower power processing cores. While some applications can readily be adapted to this change in computer architecture, a large class of applications dealing with unstructured data poses significant challenges to achieving performance on these new architectures. In this thesis we propose new algorithms and methods for the problems of graph partitioning, fill reducing ordering, and graph clustering on multicore architectures. These problems are used for the discovery and introduction of structure into this data. This is important in the fields of scientific and parallel computing, data sciences, life sciences, and integrated circuit design.

Our new methods for graph partitioning exploit current multicore architectures to greatly reduce both runtime and memory requirements. Our methods rely on a model of coarse grained parallelism while effectively maximizing data locality. We develop new methods for each phase of multilevel graph partitioning including a new aggregation method, an adaptive parallel formulation of initial partitioning, a high speed method for performing greedy refinement, and a new method for high quality refinement which is highly scalable. These new algorithms result in 3× lower runtimes than previous methods, while matching them in terms of quality using the greedy refinement method. Our new high quality and highly scalable refinement method improves the solution quality by 8%.

We develop new methods for generating fill reducing orderings of sparse matrices. This includes a new method of vertex separator refinement which manages to achieve effective parallelization while still matching the quality of the best serial algorithms. We introduce a task scheduling method specifically for nested dissection and show that it results in significantly better performance over the task schedulers available in current threading libraries. Overall, our algorithms for nested dissection are 1.5× faster than existing parallel methods and reduce the fill-in by 3.7% and operation count by 14% of Cholesky factorization compared to the orderings produced by other parallel methods.

This matches the quality of orderings generated by serial methods while achieving a 10.1× speedup on 16 cores.

Finally, we develop new serial and parallel algorithms for the graph clustering problem. We take many of the strategies we used for parallelizing graph partitioning on multicore architectures, and adapt them to the modularity maximization problem. We develop new methods for vertex aggregation, initial clustering, and cluster refinement. Our serial algorithms are an order of magnitude faster than state of the art serial methods while producing clusterings of equal or greater quality. Our parallel algorithms are $4.5 - 27.2\times$ faster than other parallel methods and achieve 8.9× speedup on 16 cores while exhibiting less than 1% degradation in solution quality compared to their counterparts. Our implementation of these algorithms can cluster a three billion edge graph in under 90 seconds on a single computer.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Achieving high computational throughputs on unstructured problems is a long standing challenge. The mutlicore era of computing has only recently arrived, and the lack of structure in these problems prevents the straight forward extraction of parallelism. Failing to extract parallelism from a problem is no longer an option as modern computer processors improve in performance by becoming increasingly parallel. In this thesis, we focus on problems that can be used to discover and introduce structure in unstructured data and computations on shared memory parallel systems. Specifically, we propose new algorithms and methods for graph partitioning, fill reducing orderings for sparse matrices, and graph clustering. These problems are key enabling technologies for efficiently operating on very large, sparse, and unstructured graphs and matrices with extensive applications. Our algorithms exploit the shared memory parallelism of multicore architectures, and result in substantial gains in terms of runtime as well as solution quality.

1.1 Problems & Applications

A common method for representing the data in unstructured problems is through a graph. That is, an object composed of vertices representing entities, and edges representing relationships between entities. For social networks, vertices are users and edges are friendships between users. For fluid dynamics simulations, vertices are flow velocities at discretized points, and edges connect physically proximal points.

1.1.1 Graph Partitioning

Graph partitioning is a technique used to decompose a graph such that connections between components are minimized and the size of each component is balanced. A common strategy for distributing work on parallel compute systems is to partition the task-dependency graph [3]. By minimizing the number of partition spanning edges, the number of dependencies crossing partition boundaries is indirectly minimized. This strategy is extensively applied to finite element analysis, which is a numerical method used in aerospace, civil, and mechanical engineering. Finite element analysis is performed by first discretizing a problem's domain into a set of elements. Calculating the next value of an element depends upon the values of the elements around it. The problem can be formulated as a large sparse system of equations which are then solved via direct or iterative methods. A good partitioning of a graph for finite element analysis minimizes the number of elements that need to be communicated between processors, which in turn minimizes the amount of time spent on communication for these parallel compute systems. In addition to minimizing the communication, a good partitioning will also result in a load balanced system, reducing the amount of time processors spend idle which reduces the total runtime.

When performing irregular computations on accelerators, the dataset often cannot fit within the memory of the accelerator. A solution is to partition the dataset into chunks that fit in the memory of the accelerator, and then schedule them for execution one after the other [4]. Each data chunk contains the data associated with the vertices in in its partition as well as the data associated with adjacent vertices in other partitions, which are required for the computation. Minimizing connectivity between partitions reduces this data duplication and improves efficiency by increasing the ratio of computation to data as well potentially reducing the total number of required partitions.

For problems dealing with irregular data that are solved via a *divide and conquer* strategy, graph partitioning plays an important role in determining the efficiency, effectiveness, and the expressible parallelism of these solutions. The speed with which the partitioning is made effects the efficiency of the *divide* step, and the quality of the partitioning effects the efficiency of the *combine* step. Graph partitioning has become an integral part of route planning systems. A common strategy [5, 6, 7] in route planning algorithms is to partition the road network and compute shortest paths between

points on the partition boundaries as a preprocessing step. Then, when a shortest path query is made, the shortest paths within the start and end partitions are found, and the precomputed inter-partition shortest path is used to join them. The number of inter-partition edges directly impacts the amount work needed in preprocessing and the amount of space the preprocessed data requires. In the generation of large Steiner Trees, graph partitioning is used to break up the graph into smaller components, for which the optimal Steiner Trees can be found [8]. These trees are then joined through boundary vertices of the partitions.

In VLSI design, graph partitioning is used to decrease design complexities by breaking up large circuits into relatively independent components and to place prototype circuits across multiple FPGAs for system emulation [9]. A good partitioning for circuit placement will not only minimize the number of connections between components, but also reduce the length of wires needed to connect components, thus reducing required signal strength [10]. This can decrease the power required by a circuit and/or the cost to manufacture it. Furthermore, it can be the difference between whether not a circuit is viable (e.g., the number of edges leaving a partition cannot exceed the number of available pins on a component).

The performance of many of these applications depends up on the speed at which the partitioning is made and the number of partition spanning edges. Existing solutions make compromises in terms of speed and parallelism, partition quality, and/or the range of graphs they can effectively partition.

1.1.2 Fill Reducing Ordering

Sparse matrices are irregular data structures that do not store the zero-valued elements which make up the majority of their entries. This makes them a powerful tool as their use can result in significant savings of storage space as well as computation. Fill reducing orderings are permutations on the input matrix which decrease the number of non-zero elements created by direct sparse methods [11]. Cholesky factorization is a direct method for factoring symmetric matrices into a lower triangular component. This factor can then be used to obtain the solution to systems of linear equations, a common step in scientific computing and computer aided design.

The factor is found in a manner similar to Guassian elimination, where rows/columns

are repeatedly eliminated. This elimination can result in fill-in, where non-zero entries are created in the factor. Orderings which result in too much fill-in can not only greatly increase the runtime, but can also cause the memory requirements to exceed the available memory of the system. Thus, we want to find orderings for which Cholesky factorization will produce as few additional non-zeros entries as possible.

1.1.3 Graph Clustering

Graph clustering is a technique for analyzing the structure of a graph by identifying groups of well-connected vertices. Modularity [12] is one of the most widely used metrics for determining the quality of non-overlapping graph clusterings, especially in the network analysis community. Discovering this structure is an important task in social network, biological network, and web analysis.

In social networks, clusters of vertices represent groups of people tied together through some form of interaction. What these groups represent depends upon what each link in the graph models. This can identify implicit common properties among people [13]. This can be researchers writing papers on the same subject, groups of individuals participating in activities together, and the classification of individuals with unknown associations, when only some associations are known. These properties can then be used in recommender systems [14] and matrix completion problems [15].

In web graphs, clustering is used to identify groups of related pages. These groupings are used for web page categorization [16] and spam detection [17].

Graph clustering is also used in the biological sciences [18]. Protein-protein interaction graphs model physical contacts between proteins. Clusters in protein-protein interaction graphs represent protein complexes and functional modules. Gene regulatory graphs model regulators and their interactions in expressing proteins and mRNA. Genes placed in the same clusters of regulatory networks tend to be functionally related. Clusterings also allow for the association diseases with genes [19]. Metabolic graphs represent the different pathways of biochemical reactions within an organism. Clustering of metabolic graphs allows for the categorization of cell components.

1.2 Emerging Challenges

The problems of graph partitioning, fill reducing ordering, and graph clustering have been well studied, and many mature solutions exist to these problems. However, as computational models continue to be applied to new domains, the input graphs/matrices to these problems have increased in both size and variety. Furthermore, computer architecture continues to evolve, and the rise of multicore architectures poses new challenges for these problems and render existing solutions inefficient.

We have recently seen the rise of Big Data, the massive amount of data being collected and generated by ubiquitous sensor deployment and digital activity monitoring. This emerging wealth of data promises to hold new scientific, medicinal, and commercial insights. A large amount of this data is sparse or unstructured, leaving it to be naturally modeled as a graph. The increasing number and size of these datasets mandates that graph partitioning, ordering, and clustering algorithms are both extremely fast and make efficient use of available memory. Furthermore, as researchers continue to pioneer new areas, they apply these partitioning, ordering, and clustering techniques to datasets with different properties than those for which existing algorithms were originally designed.

The speed of individual computer processors have largely plateaued and in some cases even decreased, while the number of processing cores per processor has gone from one to many. For applications to take advantage of the increased capabilities offered by modern processors, they need to be able to efficiently execute on all processing cores concurrently. Because improvements in memory bandwidth and latency have been made much slower than the increase in the compute power of processors, performance can only be achieved by re-using data while it is on the processor.

Several algorithms have been developed for distributed memory parallel systems for graph partitioning and fill reducing ordering [20, 21]. While these algorithms work well when each processor has its own memory system, their execution on modern multicore systems result in large degrees of memory contention and duplication, and see limited benefit as core counts continue to increase. This poses a significant problem for fields which depend on the processing and analysis of unstructured data, as it limits the size of the data they can use.

1.3 Contributions

The contributions of this thesis are the development of effective and efficient shared memory parallel algorithms for graph partitioning, ordering, and clustering. We show that on multicore architectures our algorithms achieve substantially better performance than state of the art methods.

1.3.1 Graph Partitioning

We develop and compare multiple approaches for parallelizing each of the three phases of multilevel graph partitioning: coarsening, initial partitioning, and uncoarsening using shared memory [22]. We develop and study new aggregation schemes which allow for the coarsening phase to achieve strong parallel scalability. We introduce effective methods for parallelizing the initial partitioning phase in which threads can selectively work cooperatively or independently to create the initial partitioning depending the number of threads and the size of the coarsest graph. We also present an efficient method for performing greedy refinement in parallel. The combination of these algorithms results in a significant performance improvement over previous serial and distributed memory methods. That is, a 13× speedup over the best serial method on a 32 core system which is over twice as fast as the best parallel method.

We further build upon these methods by developing algorithmic improvements and optimizations to both reduce runtime and improve the robustness of previous algorithms with respect to the larger range of graphs [23]. These changes range from implementation level optimizations such as software prefetching, to high level algorithmic changes which allow for the efficient partitioning of graphs with skewed degree distributions. These improvements allow for a further 2× reduction in runtime.

We also propose a new method for performing high quality refinement of partitioning in parallel [24]. This method is capable of breaking out of local minima in terms of the number of edges spanning partitions, something which was previously unachieved for a parallel refinement method. Our new algorithm scales well, up to 16.7× on a 24 core system, while matching the quality of the best serial algorithms.

1.3.2 Fill Reducing Ordering

We develop high-performance shared memory parallel algorithms for generating fill reducing orderings [25] via nested dissection. In nested dissection [26, 27], balanced minimum vertex separators are recursively found in the graph representing the non-zero pattern of the sparse matrix, and are used to reorder the rows and columns. This restricts the location of new non-zero entries, and the smaller the separator, the fewer non-zero entries will be created.

We develop algorithms for finding high quality vertex separators in parallel by building on the shared memory parallel work in this thesis. To achieve both speed and quality, we develop a parallel refinement method which is able to break out of local minima. This new method works by selecting several independent subgraphs along the separator and using state of the art serial refinement concurrently on each subgraph. Then, a parallel greedy refinement strategy is applied to reduce the parts of the separator not included in these subgraphs. This results in separators that are over 8% smaller than either approach on its own. This allows the first several levels of vertex separators to be found in parallel. To ensure strong performance at the higher levels of nested dissection, we develop and analyze a parallel task scheduling scheme specifically for the nested dissection problem.

We show that our parallel methods for nested dissection are 1.5× faster than current parallel methods and 10.1× faster than the best serial method on a 16 core system. Our method produces ordering which result in 3.7% less fill-in and require 14.0% fewer operations when performing Cholesky decomposition than other parallel methods. This matches the quality of orderings produced by serial methods.

1.3.3 Graph Clustering

We apply the multilevel paradigm to the modularity graph clustering problem [28]. We improve upon the state of the art by introducing new efficient methods for coarsening graphs, creating initial clusterings, and performing local refinement on the resulting clusterings. These serial algorithms are over 5.6× faster than state of the art serial methods and produce results of equal or greater quality.

We develop shared-memory parallel formulations of these algorithms to take full

advantage of modern architectures. We propose a new method of contracting more than two vertices together in parallel, which is necessary to achieve both performance and quality on the graph clustering problem. We develop a technique which allows for the concurrent updating of total cluster internal and external edges, as required by the modularity objective. Our parallel algorithms are 4.5–27.2× faster than current state of the art parallel methods. Our parallel algorithms exhibit significant parallel speedup, up to 8.9× on 16 cores, with less than one percent degradation of clustering quality, achieving the highest quality among parallel methods. Our algorithms work well on large graphs, clustering a graph with over 105 million vertices and 3.3 billion edges in 90 seconds.

1.4 Outline

This thesis is organized as follows:

- In Chapter 2 we define the notation used for graphs and formally define the problems addressed.
- In Chapter 3 we review the datasets and their properties used in this thesis as well as multicore architectures.
- In Chapter 4 we present an overview of prior work done on the subjects of graph partitioning, fill reducing ordering, and graph clustering.
- In Chapter 5 we present our work on developing algorithms for shared memory parallel graph partitioning.
- In Chapter 6 we improve upon these graph partitioning methods to overcome new challenges in the way of hardware as well as input graphs.
- In Chapter 7 we present a new high quality shared memory parallel method of partition refinement.
- In Chapter 8 we present high-performance shared memory parallel methods for the nested dissection problem.

- In Chapter 9 we present high-performance serial and shared memory parallel algorithms for the modularity graph clustering problem.
- In Chapter 10, we discuss the collective impact of the works presented in this thesis, and discuss future directions.

1.5 Related Publications

- **Dominique LaSalle** and George Karypis. Multi-threaded graph partitioning. *In Parallel & Distributed Processing Symposium (IPDPS), 2013 IEEE 27th International*. IEEE, 2013.
- **Dominique LaSalle** and George Karypis. Multi-threaded modularity based graph clustering using the multilevel paradigm. *Journal of Parallel and Distributed Computing*, volume 76, pages 66–90, 2015.
- **Dominique LaSalle** and George Karypis. Efficient nested dissection for multicore architectures. *Euro-Par 2015: Parallel Processing*, pages 467–478. Springer Berlin Heidelberg, 2015.
- **Dominique LaSalle**, Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, George Karypis, and Pradeep Dubey. Improving graph partitioning for modern graphs and architectures. Proceedings of the Fifth Workshop on Irregular Applications - Architectures and Algorithms, IAˆ3 2015.
- **Dominique LaSalle** and George Karypis. A parallel hill-climbing algorithm for graph partitioning. Technical Report 15-019, University of Minnesota, 2015.

Chapter 2

Definitions & Notation

A simple undirected *graph* $\boldsymbol{G(V, E)}$ consists of a set of vertices V and a set of edges E, where each edge $e = \{v, u\}$ is composed of an unordered pair of vertices (i.e., $v, u \in V$). The number of vertices is denoted by the scalar $\boldsymbol{n} = |V|$, and the number of edges is denoted similarly as $\boldsymbol{m} = |E|$. Each vertex $v \in V$ can have a positive weight associated with it denoted by $\boldsymbol{\eta(v)}$. Let $\eta(U)$ also denote the total weight of a set of vertices U ($\eta(U) = \sum_{v \in U} \eta(v)$). Each edge $e \in E$ can have a positive weight associated with it and is denoted by $\boldsymbol{\theta(e)}$. If there are no weights associated with the edges or vertices, then their weights are assumed to be one.

Given a vertex $v \in V$, its set of adjacent vertices is denoted by $\boldsymbol{\Gamma(v)}$ and is referred to as the *neighborhood* of v. For an unweighted graph, $\boldsymbol{d(v)}$ denotes the degree of the vertex v which is the number of edges incident to v (e.g., $d(v) = |\Gamma(v)|$), and for the case of edge weights, $d(v)$ denotes the total weight of edges incident to v (e.g., $d(v) = \sum_{u \in \Gamma(v)} \theta(\{v, u\})$).

A *partition* $\boldsymbol{P_i}$ (or *cluster* C_i) is a subset of vertices in the graph, $P_i \subset V$. In order to facilitate discussions about the effects of moving vertices between partitions, let $\boldsymbol{d_{int}(v)}$ denote the internal degree of v, that is, the sum of the weight of edges connecting v to the partition in which it resides. Let $\boldsymbol{d_{ext}(v)}$ denote the external degree of v, that is, the sum of the weight of edges connecting v to partitions other than the one in which it resides. Let $\boldsymbol{d_{P_i}(v)}$ denote the sum of the weight of edges connecting the vertex v to the partition P_i. Finally, let $\boldsymbol{\pi(v)}$ denote the number of external partitions to which v is connected.

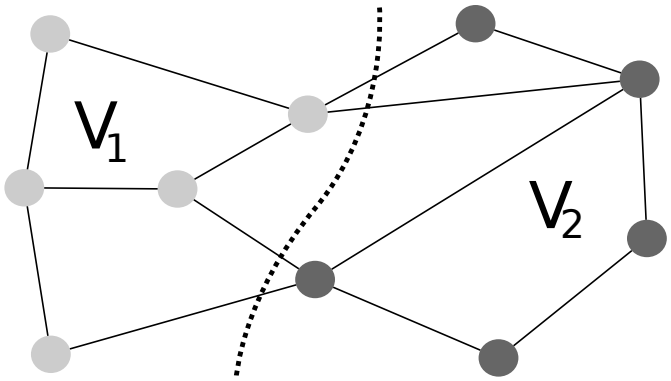


Figure 2.1: A two-way partitioning with an edgecut of four.

The sum of vertex degrees within a partition is denoted as $\boldsymbol{d(P_i)}$ (i.e., $d(P_i) = \sum_{v \in P_i} d(v)$). The internal degree $\boldsymbol{d_{int}(P_i)}$ of a partition P_i is the number of edges (or sum of the edge weight) that connect vertices in P_i to other vertices within P_i. The external degree $\boldsymbol{d_{ext}(P_i)}$ of a partition P_i is the number of edges (or sum of the edge weight) that connect vertices in P_i to vertices in other partitions. The neighborhood of a partition P_i, denoted by $\Gamma(P_i)$, is the set of all partitions connected to P_i by at least one edge. The number of edges connecting the partition P_i to P_j is denoted as $d_{P_j}(P_i)$. Since G is an undirected graph, $d_{P_j}(P_i) = d_{P_i}(P_j)$. Similarly, the number of edges (or total edge weight) connecting a vertex v to the partition P_i is denoted as $\boldsymbol{d_{P_i}(v)}$ (i.e., $d_{P_i}(v) = \sum_{u \in P_i \cap \Gamma(v)} \theta(\{v, u\})$. We will denote the partition P_i with the vertex v removed, as $\boldsymbol{P_i - \{v\}}$, and the partition P_j with the vertex v added as $\boldsymbol{P_j + \{v\}}$.

When discussing parallel complexities, we will refer to the number of threads/processes being used as $\boldsymbol{p}$.

2.1 Graph Partitioning

The graph partitioning problem takes as input a simple undirected graph G, and divides the vertex set into disjoint subsets of vertices (partitions), $V = P_1 \cup \ldots \cup P_k$. We will refer to this collection of partitions as a *partitioning* $\boldsymbol{P} = \{P_1, \ldots, P_k\}$ and $\boldsymbol{k}$ as the number of partitions (said to be a k-way partitioning).

In a partitioning, the edges crossing from one partition to another form an *edge*

separator. The number of edges in the separator (or total edge weight) is known as the *edgecut*:

$$edgecut = \sum_{i=1}^{k} \sum_{v \in P_i} \sum_{u \in \Gamma(v), u \notin P_i} \theta(\{v, u\}).$$

A balanced two-way partitioning with an edgecut of four is shown in Figure 2.1. The dotted line crosses the four edges that make up the edge separator, and divides the vertices into two partitions.

The objective of the graph partitioning problem is to find a partitioning that minimizes the edgecut while satisfying the constraint that the size of partitions are balanced within some tolerance $\boldsymbol{\epsilon}$. That is,

$$k \frac{\max_i \eta(P_i)}{\eta(V)} \leq 1 + \epsilon.$$

This ensures that all partitions are nearly the same size (or exactly the same size where $\epsilon = 0$). It can be generalized to partitions of non-uniform size:

$$\max_i \frac{\eta(P_i)}{w_i} \leq 1 + \epsilon,$$

where w_i is the target size for partition P_i.

The balanced graph partitioning problem is known to be NP-complete [29], and finding solutions within some approximation factor has been shown to be NP-complete as well [30]. The balanced graph partitioning problem can be reduced to a max-cut problem [31] in order to show that it is NP-complete.

2.1.1 Vertex Separators

A *vertex separator* $\boldsymbol{S}$ is a subset of the vertices in the graph, $S \subset V$, which divides the graph into two partitions, $\boldsymbol{A}$ and $\boldsymbol{B}$. That is, every path from a vertex in A to a vertex in B (or vice versa) contains at least one vertex from S. Figure 2.2 shows a balanced vertex separator of size two. The size of a vertex separator is the sum of the weight of the vertices in the separator $\sum_{v \in S} \eta(v)$. The balance constraint of vertex separators is only with respect to the vertices in A and B, but not those within S:

$$2 \frac{\max(\eta(A), \eta(B))}{\eta(A) + \eta(B)} \leq 1 + \epsilon.$$

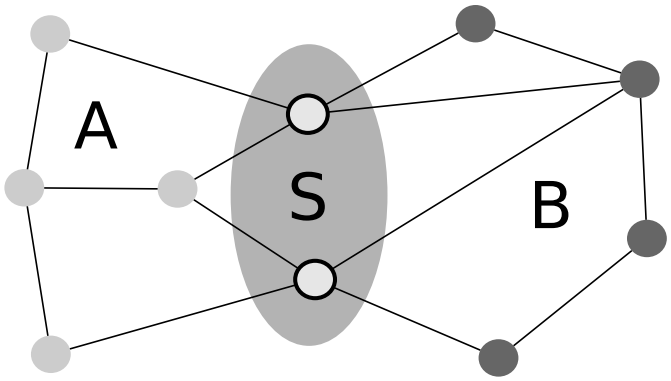


Figure 2.2: A vertex separator of size two.

2.2 Fill Reducing Ordering

Performing addition/subtraction on a sparse matrix, as is done in many direct methods, can result in the creation of non-zero entries in a sparse matrix. These created entries are referred to as *fill-in*. This can lead to significant increases in memory and computational requirements.

Eliminating a row/column pair from a symmetric sparse matrix as done in Cholesky decomposition involves performing a rank-1 update to the rest of the matrix. An *elimination graph* is a model for representing the effect of this update on the non-zero pattern of the matrix. When a vertex v is eliminated, all edges incident to v are removed, and edges are added between all neighbors of v. These added edges correspond to non-zeros being added to the sparse matrix. The order in which vertices are eliminated will impact the number of edges created. Any neighbors of v eliminated prior to it, will not be connected as a result of v's elimination. For any pair of v's neighbors already connected by an edge, v's elimination will not result in an edge being added between them. A fill reducing ordering of the matrix will increase the occurrences of these situations when the rows/columns are eliminated in that order.

2.3 Graph Clustering

A *clustering* of a graph is a division of its vertex set V into disjoint subsets, $\boldsymbol{C} = \{C_1, \ldots, C_k\}$, such that connectivity within clusters is maximized. Clustering differs

from partitioning in two ways. First, it does not have a balance constraint. Instead, the balance of the size/weight of the clusters is part of the objective function. Second, the number of clusters k is an output parameter rather than an input parameter. That is, it is up to the clustering algorithm to select the best k for the graph such that the objective is maximized. We will use the same notation to denote connectivity between clusters and vertices as we did for partitions (i.e., $d_{C_i}(v) = \sum_{u \in C_i \cap \Gamma(v)} \theta(\{v, u\})$.

The clustering objective focused on by this thesis is *modularity* [12], which is an effective means for identifying clusters of vertices within a graph and has become ubiquitous in recent graph clustering/community detection literature. Modularity measures the difference between the expected number of intra-cluster edges and the actual number of intra-cluster edges. Denoted by $\boldsymbol{Q}$, the modularity of a clustering C is expressed as

$$Q = \frac{1}{d(V)} \left(\sum_{C_i \in C} \left(d_{int}(C_i) - \frac{d(C_i)^2}{d(V)} \right) \right),$$

where $d(V)$ is the total degree of the entire graph (i.e., $d(V) = \sum_{v \in V} d(v)$). From this, we can see the modularity Q_{C_i} contributed by cluster C_i is

$$Q_{C_i} = \frac{1}{d(V)} \left(d_{int}(C_i) - \frac{d(C_i)^2}{d(V)} \right).$$

The value of Q ranges from -0.5, where all of the edges in the graph are inter-cluster edges, and approaches 1.0 if all edges in the graph are intra-cluster edges and there is a large number of clusters. Note that this metric does not use the number of vertices within a cluster, but rather only the edges. Subsequently, vertices of degree zero, can arbitrarily be placed in any cluster without changing the modularity. Maximizing modularity is an NP-complete problem [32], as it can be reduced to a 3-partition problem [33].

Chapter 3

Background

3.1 Graphs

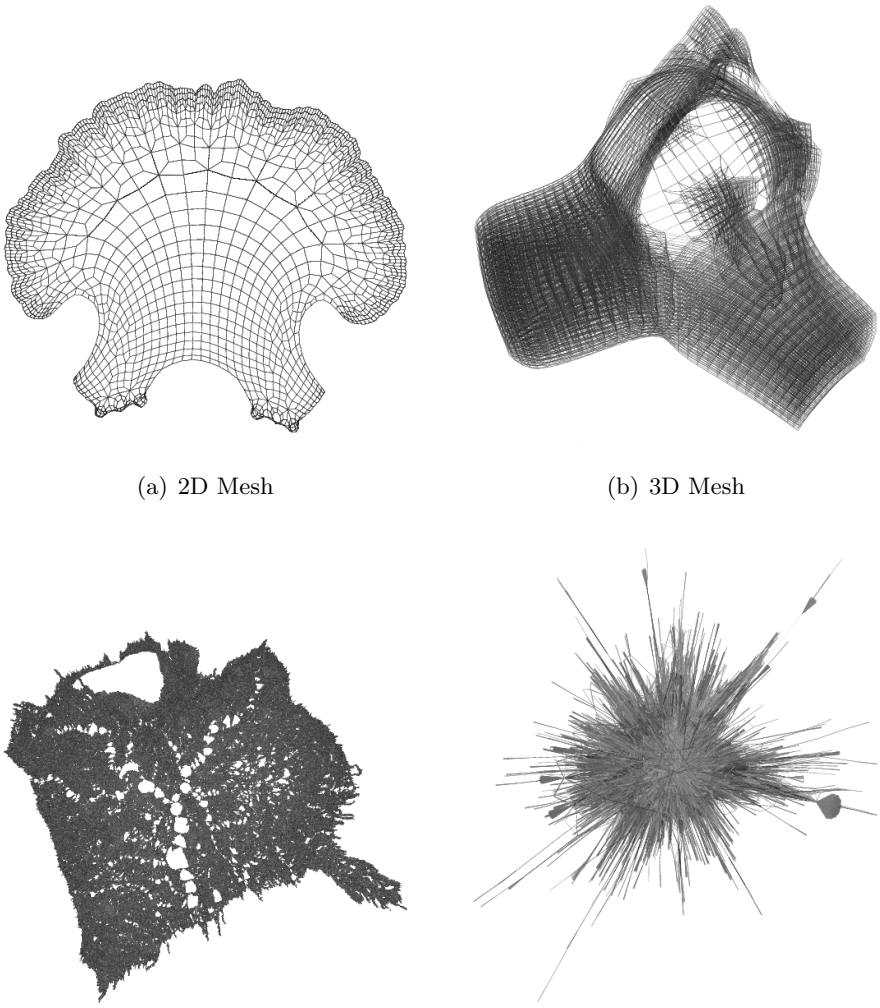
Graphs vary in several important properties depending on their domain of origin. Figure 3.1 shows four types of graphs commonly used in computational problems.

The *diameter* of a graph is the length of the longest shortest path. That is, a graph has a diameter of d, if for every pair of vertices in the graph u and v, the length of the shortest path is less than or equal to d. The diameter of a graph is an important property of the graph's structure. High diameter graphs tend to have low maximum degree. Low diameter graphs, tend to either have very high average degree and/or high maximum degree.

The vertex *degree distribution* and average vertex degree describe the connectivity of the graph. A graph with a relatively uniform degree distribution (i.e., the maximum vertex degree is within some small multiple of the average degree), will tend to have a large diameter for its size. Conversely, a graph with a skewed degree distribution (i.e., the maximum vertex degree is significantly larger than the average degree) will tend to have a small diameter.

3.1.1 Meshes

The graphs used in scientific computing are often derived from *meshes*, which are the discretization of 2D or 3D problem spaces into polygonal or polyhedral elements respectively. Depending on the operation, the graph representing the computation to be done



(a) 2D Mesh

(b) 3D Mesh

(c) Road Network

(d) Social Network

Figure 3.1: Various graph types with differing properties: (a) a two dimensional finite element mesh, (b) a three dimensional finite element mesh, (c) a road network, and (d) a social network [1, 2].

is either the mesh itself where the corners of the elements are vertices and the edges of the elements are edges in the graph, or where the graph is the dual of the mesh where each element is a vertex and elements sharing an edge/face are connected by an edge in the graph. Meshes and their duals have relatively uniform degree distributions and

Table 3.1: 2D Mesh graphs used in this thesis.

Graph	Vertices	Edges	Max. Deg.	Avg. Deg.
`t60k` [34]	60,005	89,440	3	3.0
`wing` [34]	62,032	121,544	4	3.9
`fe_ocean` [34]	143,437	409,593	6	5.7
`333SP` [35]	3,712,815	11,108,633	28	6.0
`AS365` [35]	3,799,275	11,368,076	14	6.0
`NLR` [35]	4,163,763	12,487,976	20	6.0
`adaptive` [36]	6,815,744	13,624,320	4	4.0

high diameters. Table 3.1 shows the 2D mesh graphs used for experiments in this thesis. Two dimensional meshes have low average degree. A 2D mesh is planar by definition, and thus can only have an average degree of at most six (strictly less than six for finite graphs)[37]. This can be seen from *Euler's polyhedron theorem* [38] (or just *Euler's theorem*):

$$n - m + f = 2, \tag{3.1}$$

and the fact that an edge can be incident to at most two faces, and a face must have at least three edges (for $m > 2$):

$$2m \geq 3f. \tag{3.2}$$

Transforming equation (3.2) into $f \leq 2m/3$, and plugging it into equation (3.1), we get:

$$n > 2 + m - \frac{2}{3}m,$$

$$3n > 6 + m. \tag{3.3}$$

Which tells us that the average degree ($2m/n$) must be less than six:

$$\frac{6m}{6+m} < 6.$$

.

Lipton and Tarjan [39] showed that a two-way vertex separator of a planar graph will contain on the order of $\sqrt{n}$ vertices. This is accomplished by combining the *Jordan curve theorem* [40], *Kuratowski's theorem* [41], and the bound on the number of edges in a planar graph equation (3.3).

Table 3.2: 3D Mesh graphs used in this thesis.

Graph	Vertices	Edges	Max. Deg.	Avg. Deg.
fe_pwt [34]	36,519	144,794	15	7.9
fe_body [34]	45,087	163,734	28	7.3
vibrobox [34]	12,328	165,250	120	26.8
bcsstk33 [34]	8,738	291,583	140	66.7
bcsstk29 [34]	13,992	302,748	70	43.3
brack2 [34]	62,631	366,559	32	11.7
fe_tooth [34]	78,136	452,591	39	11.6
bcsstk31 [34]	35,588	572,914	188	32.2
fe_rotor [34]	99,617	662,431	125	13.3
598a [34]	110,971	741,934	26	13.2
bcsstk32 [34]	44,609	985,046	215	44.2
bcsstk30 [34]	28,924	1,007,284	218	69.7
wave [34]	156,317	1,059,331	44	13.6
144 [34]	144,649	1,074,393	26	14.9
m14b [34]	214,765	1,679,018	40	15.6
auto [34]	448,695	3,314,611	37	14.8
med_fe	1,752,854	20,552,976	230	23.6
ldoor [1]	952,203	22,785,136	76	47.6
hd2_fe	1,118,496	31,255,782	99	55.9
Serena [42]	1,391,349	31,570,176	248	55.1
audikw1 [1]	943,695	38,354,076	344	81.3
channel-500x [43]	4,802,000	42,681,372	18	17.8
dielFilterV3 [44]	1,102,824	44,101,598	269	63.4
Flan_1565 [42]	1,564,794	57,920,625	80	74.0
large_fe	7,221,643	83,149,197	60,853	23.0

Table 3.2 shows the 3D mesh graphs used for experiments in this thesis. Three dimensional meshes have significantly higher average and maximum degree than two dimensional meshes. Unlike 2D meshes, 3D meshes do not have a bound on their average degree, and many represent the highest average degree of any graph used in

Table 3.3: Road graphs used in this thesis.

Graph	Vertices	Edges	Max. Deg.	Mean Deg.
`luxembourg.osm` [48]	114,599	119,666	6	2.1
`belgium.osm` [48]	1,441,295	1,549,970	10	2.2
`asia.osm` [48]	11,950,757	12,711,603	9	2.1
`road_usa` [48]	23,947,347	28,854,312	9	2.4
`europe.osm` [48]	50,912,018	54,054,660	13	2.1

this thesis. As with 2D meshes, the maximum degree is not significantly higher than the average degree. The only exception to this is the `large_fe` graph, which has a maximum degree three orders of magnitude larger than its average degree. High degree vertices can be used to couple equation components together in nodal meshes, and in dual meshes high degree vertices are known as *super-elements* [45, 46]. Super-elements often represent a simplified structure in order to reduce computation.

Three dimensional meshes also tend to have higher diameters. Miller et al. [47] showed that for D-dimensional graphs with certain properties common to finite element meshes, a vertex separator of size $O(n^{1-1/D})$ exists. For 3D meshes this implies that size of an edge separator will tend to be on the order of $n^{2/3}$.

3.1.2 Road Networks

Table 3.3 shows the graphs of road networks used for experiments in this thesis. As can be seen from the average degree, road networks tend to be very sparse. They also tend to have very high diameters, which is a result of their near planarity and low maximum degree. Road networks tend to have areas of low connectivity, which are caused by natural barriers such as rivers, mountains, and bodies of water where few if any roads cross.

3.1.3 Social Networks

Table 3.4 shows the social network graphs used for experiments in this thesis. Social networks tend to have skewed degree distributions. However, how skewed the distribution is depends upon the network. For example, the vertex with the highest degree in

Table 3.4: Social network graphs used in this thesis.

Graph	Vertices	Edges	Max. Deg.	Mean Deg.
`email` [48]	1,133	5,451	71	9.6
`polblogs` [48]	1,490	16,715	361	22.4
`PGPgiantcompo` [49]	10,680	24,316	205	4.6
`astro-ph` [48]	16,706	121,251	360	14.5
`cond-mat-2005` [50]	40,421	175,691	278	8.7
`flickr` [51]	820,878	6,625,280	10,891	16.1
`citationCiteseer` [48]	268,495	1,156,647	1,318	8.6
`coAuthorsCiteseer` [48]	227,320	814,134	1,372	7.2
`coPapersDBLP` [48]	299,067	977,676	336	6.5
`cit-Patents` [52]	3,774,768	16,518,947	793	8.8
`soc-pokec` [53]	1,632,803	22,301,964	14,854	27.3
`soc-LiveJournal1` [54]	4,847,571	42,851,237	20,333	17.7
`com-orkut` [55]	3,072,441	117,185,083	33,313	76.3
`com-friendster` [55]	65,608,366	1,806,067,135	5,214	55.1

the `flickr` graph connects to 1.3% of the vertices. However, the vertex with the highest degree in the `cit-Patents` graph connects to only 0.02% of the vertices. In fact, we can see that for the online social networks, the maximum degree vertex connects to a smaller fraction of the vertices as the network size increases. Social networks tend to have less strong community structures compared to road networks and web graphs [28], as the clusters of vertices also tend to be highly connected to other clusters.

Whang et al. [56] showed that many social networks can be described by their high-degree vertices which form the *skeleton* of the network. This contributes to the networks small diameter, as most vertices are within one hop of this skeleton, thus very few hops are needed reach other vertices which are not part of the skeleton.

3.1.4 Web Graphs

Table 3.5 shows the web graphs used for experiments in this thesis. While in practice web graphs are directed networks (i.e., links between pages are unidirectional), for the

Table 3.5: Web graphs used in this thesis.

Graph	Vertices	Edges	Max. Deg.	Mean Deg.
`in-2004` [57]	1,382,908	13,591,473	21,869	19.6
`eu-2005` [57]	862,664	16,138,468	68,963	37.4
`wikipedia-2007.` [58]	3,566,908	42,375,912	187,671	23.8
`uk-2002` [57]	18,520,486	261,787,258	194,955	28.3
`uk-2007-05` [48]	105,896,555	3,301,876,564	975,419	21.8

Table 3.6: Synthetic graphs used in this thesis.

Graph	Vertices	Edges	Max. Deg.	Mean Deg.
`preferentialAttachment` [48]	100,000	499,985	983	10.0
`smallworld` [48]	100,000	499,998	17	10.0
`G_n_pin_pout` [48]	100,000	501,198	25	10.0
`rgg_n_2_17_s0` [48]	131,072	728,753	28	11.1
`hugetrace-00020` [60]	16,002,413	23,998,813	3	3.0
`delaunay_n24` [48]	16,777,216	50,331,601	26	6.0

graph problems in this thesis, we consider the versions undirected of these graphs.

Web graphs have degree distributions further skewed than that of social networks. This is largely due to their directed nature. For example, the Wikipedia page *Animal* has $107,424$ incoming links [59]. Web graphs also tend to have very strong community structure [28].

3.1.5 Synthetic Graphs

There are many ways to generate synthetic graphs, depending on the properties desired. One of the most common methods of synthetic graph generation is using the Erdős-Rényi model [61], where a graph is constructed via connecting pairs of vertices randomly. This model, $G(n, p)$, takes n as the number of vertices to include and p as the probability that any two vertices are connected via an edge. If p is equal 0.5, all possible graphs with n vertices will have an equal probability of being generated. If p less than 0.5, the generated graph will tend to be sparser, whereas when p is greater than 0.5, the

generated graph will tend to be denser.

A Delaunay [62] triangulation of a set of points in two-dimensional space is set of edges connecting those points such that each internal face has only three sides, and for each three sided face, the circle going through all three points of the face contains no other points within it. Current state of the art methods for generating Delaunay triangulations use a divide and conquer approach [63] running in $O(n \log n)$ time. These methods can be extended to higher dimensions [64] (e.g., generate tetrahedrons using spheres in 3D space). Such triangulations are often used to generate the graph for finite element analysis.

Gilbert proposed the *Random Geometric Graph* [65] (RGG) model. This model works by randomly placing points on a plan using a Poisson process with a density D. Then, for every pair of points with a distance less than R, an edge is added between them. While RGG graphs are not necessarily planar, they have similar properties such as low degree, relatively uniform degree distributions, and large diameters (as a function of R).

The *Preferential Attachment* [66] notion for graphs is that when an edge is added, it is more likely to be incident to high degree vertices (i.e., high degree vertices tend to become higher degree, and low degree vertices tend to remain low degree vertices). The Barabási-Albert [67, 68] model uses this notion to construct a random graph. This model start with an initial number of connected vertices n_0, and then a new vertex is connected to each existing vertex i with probability $p_i = d(i)/2m_j$ where m_j is the number of edges present in the graph before adding the new vertex. This mean each new vertex is connected to existing vertices with probability equal to the fraction of existing edges to which they are incident. This has been shown to be a good approximation of the growth of social networks and web graphs [69].

Table 3.6 shows the synthetic graphs we used in experiments in this thesis. We use synthetic graphs which mimic many real world graph properties. The `preferentialAttachement` graph has many properties similar to web graphs. The `smallworld` and `G_n_pin_pout` graphs have many properties in common with social networks. The `rgg_n_2_17_s0`, `hugetrace-00020`, and `delaunay_n24` graphs mimic the structure of 2D and 3D meshes.

Table 3.7: Graphs of various other types used in this thesis.

Graph	Vertices	Edges	Max. Deg.	Mean Deg.
`celegans_metabolic` [70]	453	2,025	237	8.9
`power` [50]	4,941	6,594	19	2.7
`as-22july06` [50]	22,963	48,436	2,390	4.2
`memplus` [1]	17,758	54,196	573	6.1
`finan512` [34]	74,752	261,120	54	7.0
`caidaRouterLevel` [48]	192,244	609,066	1,071	6.3
`cage15` [71]	5,154,859	47,022,346	46	18.2
`vlsi_crct`	49,375,363	76,768,132	44	3.1
`nlpkkt240` [72]	27,993,600	373,239,376	27	26.7

3.1.6 Other Types of Graphs

Table 3.7 shows graphs from various other domains. The `celegans_metabolic` and `cage15` graphs are from the life sciences domain: a metabolic network and a model of DNA electrophoresis respectively. The `power` and `as-22july06` graphs are physical networks: the US power grid and internet routers for autonomous systems respectively. The graphs `memplus` and `vlsi_crct` are both circuits from the VLSI domain. The `finan512` graph is an economics problem of portfolio optimization. The `nlpkkt240` graph is a non-linear programming problem.

3.2 Parallel Architectures

In recent years we have seen everything from the nodes of high-end distributed systems to commodity workstations shift from being single-core/single-processor machines to multicore/multi-processor shared memory machines. This shift has also caused the amount of available memory per processing core to decrease [73]. Multicore architectures allow for the reduction of the amount of data that needs to be replicated between processors as well as allows for fine-grain communication and synchronization.

Figure 3.2 shows the standard layout of a modern compute node/workstation. It will have one or more processors each with one or more cores, and will have one or more

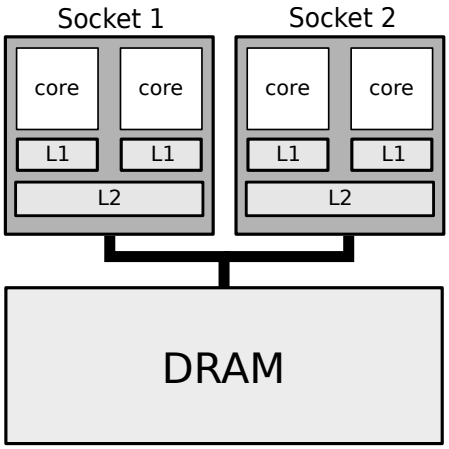


Figure 3.2: A simple layout of a multicore compute node, with two processors, each with two cores and two levels of cache.

memory banks (DRAM) which all processors can directly access. The multiple levels of cache (L1 and L2) on top of the main memory result in non-uniform access times for different memory locations (NUMA). This means that while all processing cores can access all memory addresses, the access time associated with any given address depends upon its physical location. Parallel programs on these architectures must maximize data locality in order to achieve strong performance.

3.2.1 Software Threads and Processes

For a program to utilize a multicore system, it must either to be made up of multiple processes, multiple threads, or a hybrid of both (multiple processes each with multiple threads). The difference between threads and processes at the software level is largely dependent on the operating system. On a multicore system, the operating system will map one or more process/thread to each compute core. For this thesis we will rely on the distinction made by the Linux [74], Windows [75], Solaris [76], and BSD [77] operating systems, referring to light-weight processes as *threads* and heavy-weight processes as *processes*.

Processes are heavy weight tasks. They have exclusive access to their own resources. The exception to this is that processes may map the same region of memory

via shared memory mechanisms used for IPC (such as `shmget()`, `mmap()`, etc.). However, these mechanisms come with restrictions and performance overheads [78, 79]. A process is made up of one or more threads.

Threads are light weight subtasks. They share the resources of the of the process to which they belong, except they each have their own control and execution stack. Thread creation and context switching between threads is significantly faster than process creation and context switching between processes [80]. Because threads share the same resources, communication and synchronization between threads is much less costly than between processes. This allows threads to implement a wider range of operations efficiently compared to traditional processes.

Chapter 4

Related Work

4.1 Graph Partitioning

Delling et al. [81] proposed what is one of the fastest methods of solving the balanced bisection problem exactly. Their method uses a branch and bound approach to reduce the amount of work required to explore the solution space. The upper bound is determined by using one of the heuristics discussed below, and the lower bound is determined by means of the min-cut/max-flow problem. The method is particularly strong on graphs with small bisections, where both the lower and upper bounds are much tighter. For example, they were able to find exact solutions on road networks for Belgium and the Netherlands, which have millions of vertices for which no exact solutions had been found before. However, for other classes of graphs, the maximum solvable size stayed in the thousands and tens of thousands of vertices. This branch and bound methodology is not applicable to finding k-way solutions.

4.1.1 Recursive Bisection

Recursive bisection can be used to generate an arbitrary k-way partitioning (where k is greater than two). First, a bisection is made such that partition A's target size is $\eta(V)/\lceil k/2 \rceil$ and partition B's target of size is $\eta(V)/\lfloor k/2 \rfloor$. To ensure that the balance constraint for the k-way partitioning is satisfied, when bisecting partition A we must

use a constraint of:

$$\epsilon_A = \frac{(1+\epsilon)\sum_{i=1}^{k_A} w_i}{\eta(A)} - 1,$$

where k_A is the number of partitions into which A needs to be divided. However, if the initial bisection makes full use of the balance constraint, the subsequent bisections on the heavy partition will be made with almost no imbalance tolerance, which can severely degrade solution quality. A solution to prevent this is to make each bisection with $\epsilon' = \epsilon_A / \log_2(k_A)$. This ensures that each bisection has some tolerance for imbalance. Recursive bisection can be applied to the case where k is not a power of two, by changing the target partition size for each bisection.

4.1.2 Spectral Graph Partitioning

Spectral graph partitioning uses the spectrum of the matrix representation of a graph to find good partitions. Spectral methods for partitioning graphs were developed by Donath and Hoffman [82], and many improvements have been made to spectral graph partitioning [83, 84, 85, 86] since then.

The Laplacian L of graph G is equal the diagonal matrix D of the degrees of the vertices, minus the adjacency matrix X, where x_{ij} is 1 where vertices v_i and vj are adjacent (connected), and 0 when they are not.

$$L = D - X$$

The degree of a vertex is the number of other vertices it is connected to by an edge. We can express D in terms of X, as $d_i = \sum_{j=1}^{n} |a_{ij}|$. Fiedler [87] showed that the vector related to the second smallest eigenvalue λ_2 of the Laplacian L, correlated well to the algebraic connectivity of the graph. Subsequently this vector is often referred to as the Fiedler vector.

The graph can then be bisected via this vector by placing vertices associated with high values into one partition, and placing the vertices associated with low values into the other partition. The Fiedler vector can be found via the *Implicitly Restarted Lanczos* algorithm [88].

Another method of finding the Fiedler vector is Multilevel Recursive Spectral Bisection (MRSB) proposed by Barnard and Simon [85]. This approach is based on multigrid

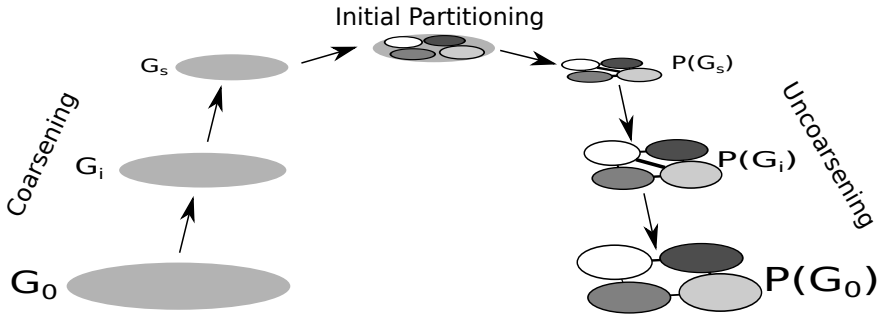


Figure 4.1: The multilevel process.

methods [89], and is similar to the multilevel methods discussed in the next section. Starting from the input Laplacian, L_0, a series of smaller/coarser approximations are generated, $L_1, \ldots, L_s$. Once the coarsest Laplacian L_s is generated, the Fiedler vector f_s of that Laplacian is found using Lanczos algorithm. Then f_s is used as an estimate of f_{s-1} based on the coarsening that took place between L_{s-1} and L_s. Because Lanczos algorithm fails to take advantage of a good approximation to the desired eigenvector, the Rayleigh Quotient Iteration is used to determine f_{i-1}.

4.1.3 Multilevel Graph Partitioning

Since their introduction over 20 years ago [90], multilevel methods for graph partitioning have become the standard approach for developing high-quality and computationally efficient solutions for graph partitioning. These algorithms solve the underlying optimization problem using a methodology that follows a *simplify & conquer* approach, initially used by multi-grid methods for solving systems of partial differential equations.

Multilevel partitioning methods consist of three distinct phases: *coarsening*, *initial partitioning*, and *uncoarsening*, as seen in Figure 4.1. In the coarsening phase, the original graph G_0 is used to generate a series of increasingly smaller or *coarser* graphs, $G_1, G_2, \ldots, G_s$. To create G_{i+1} from G_i, vertices are first *aggregated* into either pairs or groups, and then the pairs/groups are *contracted* together. In the initial partitioning phase, a partitioning of the coarsest graph G_s is generated using a direct partitioning method. Finally, in the uncoarsening phase, the initial partitioning is used to derive partitionings of the successive larger or *finer* graphs. This is done by first *projecting*

the partition of G_{i+1} to G_i, followed by partitioning *refinement* whose goal is to reduce the edgecut by moving vertices among the partitions. Since the successive finer graphs contain more degrees of freedom, such refinement is often feasible and leads to dramatic edgecut reductions.

The overall effectiveness of the multilevel paradigm depends on the approaches used to identify the set of all vertices that will be contracted during the coarsening phase and the refinement during the uncoarsening phase. Over the years various approaches have been developed and extensively evaluated [91]. For example, coarsening is usually performed by computing a matching [90, 92] or grouping [93] though approaches based on weighted aggregation have also been explored [94, 95]. The refinement is often performed using local search methods based on the Kernighan-Lin [96], Fiduccia-Mattheyses [97], or Greedy [92] refinement algorithms.

Aggregation Schemes

Karypis and Kumar [98] proposed the Heavy Edge Matching (HEM) aggregation scheme. This scheme works by having each vertex select its heaviest incident edge for matching. This is an extremely fast method for finding a maximal matching but makes no guarantees in terms of matched edge weight. Vertices are visited in increasing order of degree so as to allow low degree vertices to select matches from their limited set of neighbors before high degree vertices select their matches.

Finding a maximum matching can be done in $O(|E|\sqrt{|V|})$ time using the algorithm by Micali and Vazirani [99]. The maximum weight matching can be found in $O(|V||E| + |V|^2 \log |V|)$ time with the algorithm proposed by Gabow [100]. The globally greedy matching approximation [101] has a running time of $O(|E| \log |V|)$, using a heap to maintain the globally heaviest edge available for matching. It guarantees that the resulting matching will contain at least 1/2 of the edge weight of the maximum weight matching.

Monien et al. [102] improved upon this by reducing the runtime to linear with respect to the number of edges in the graph. This is accomplished by always adding the locally heaviest edge to the matching, rather than the globally heaviest. This still guarantees the 1/2 approximation factor. It can be shown that the matched edge set resulting from locally heaviest edge matching will contain the same edges as the set resulting

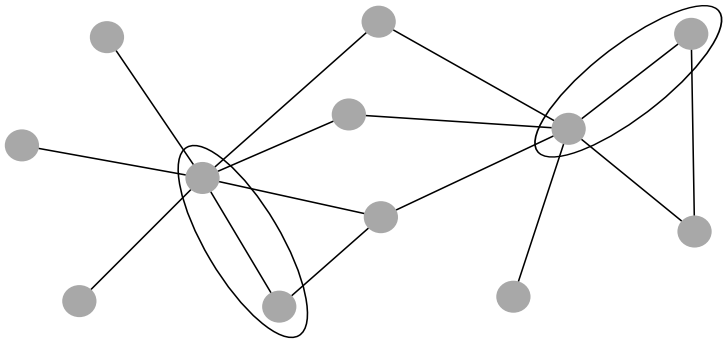


Figure 4.2: A small maximal matching.

from globally heaviest edge matching.

Algebraic distance has also been investigated as a criteria for selecting vertices to match [103, 104]. This uses Jacobian Over-Relaxation on the graph Laplacian to reweight edges to better represent the connectivity between vertices. That is, edges between vertices in the same highly connected region of the graph will tend to be given high weights, and edges between vertices in different regions of connectivity will be given low weights. This is similar to a method explored by Spielman et al. [105], for removing edges from a graph while trying to minimize the effects on the graph's structure. Chevalier and Safro [106] experimented with using weighted aggregation (WAG) schemes, whereby a fine vertex may be subdivided among coarse vertices. This minimizes the consequences of *bad* contractions (i.e., the contraction of an edge desirable for cutting) but requires more computation than standard matching methods.

A graph with a skewed vertex degree distribution has many vertices of low degree and only a few vertices of high degree. These graphs often have small maximal matchings, as shown in Figure 4.2. Other properties of these graphs include small diameters (the distance of the maximum length shortest path between any two vertices in the graph), as shortest paths will often go through these high degree vertices.

If we only find a small maximal matching during aggregation, the size of the next coarser graph will reduce by only a small faction, and G_{i+1} will be nearly the same size as G_i. Unless coarsening is terminated early, this could cause a runtime of $O(n^2+nm)$ as we would need $O(n)$ levels of coarsening, each of which takes $O(n+m)$ time. Furthermore, this would also likely lead to very uneven vertex weights in the coarsest graph, as high

degree vertices would be aggregated at each level and be of large weight, whereas low degree vertices would be unlikely to have been aggregated and be of low weight. This greatly restricts creating a balanced initial partitioning, and often leads to very poor quality and/or unbalanced partitions.

For graphs with a high diameter, the edge density of the input graph G_0 tends to be of greater than or equal density of the coarsest graph G_s. The clusters of vertices (groups of highly inter-connected vertices) are sparsely connected in these graphs, which is what causes them to have high diameters. As a result, once the clusters get contracted, only the few inter-cluster edges are left exposed. However, for graphs with small diameters, the density of G_s increases. For the diameter to be small, the longest-shortest path can only pass through a small number of clusters. Thus, the interconnection of the clusters must be relatively dense, and the number of exposed edges in G_s must be high. This means that the amount of computation associated with creating a partitioning for G_s is dramatically higher, and can exceed that of coarsening and uncoarsening.

Abou-Rjeili and Karypis [107] studied several techniques for aggregating more than two vertices together per level to overcome the reduced size of maximal matchings in graphs with skewed degree distributions. Meyerhenke et al. [108] used sized-constrained label propagation to select small clusters of vertices to aggregate together per level. They showed that this could lead to greatly reduced runtime as well as improved partition quality.

Initial Partitioning Schemes

Hendrickson and Leland [90] use spectral bisection to generate the initial partitioning. Karypis and Kumar [98] perform a size constrained breadth first search, to create a rough initial bisection, and then apply refinement to improve its quality. Walshaw and Cross [109] contract the graph to the point where the number of coarse vertices is equal to the number of partitions (k). Balance issues resulting from contraction are then handled in uncoarsening via fine grain vertex movement. When the coarsest graph is of a sufficiently small size and is sufficiently sparse, a branch and bound method [81] can used to find the minimum bisection.

Refinement Schemes

Refinement can have a large impact on the quality of a solution generated via the multilevel paradigm. Refinement techniques range from light weight greedy approaches [110], to more intensive flow-based techniques [111]. While the multilevel paradigm has been shown to produce solutions of good quality as result of the contracted edge weight [98], refinement techniques capable of breaking out of local minima offer a means to explore a wider range of solutions and improve quality.

The Greedy [110] refinement algorithm, is an extremely fast method for converging on a local minima in the edgecut of a k-way partitioning. The Greedy algorithm works by making several iterations over the boundary of the partitioning until no improvement is made in an iteration, or a maximum number of iterations have been performed. In each iteration, vertices are moved individually in greedy order to reduce the edgecut with the restriction that each vertex can move only once per iteration. Each iteration works as follows. First, all of the boundary vertices, those with edges connecting them to the opposing partition, are inserted into a priority queue. The gain associated with moving a vertex is used as the priority. This gain for the vertex v is the sum of the weight of the edges connecting it to the opposing partition minus the sum of the weight of the edges connecting it to the partition in which it resides:

$$gain = d_{ext}(v) - d_{int}(v).$$

Vertices are extracted from this queue and are considered for moving. If the gain associated with a moving a vertex is positive, and moving the vertex would not violate the balance constraint, it is moved to the opposing partition, and its neighbors are updated in the priority queue. A simple optimization for this algorithm is to only insert vertices with positive gain to into the priority queue, as they are the only ones considered for moving. Despite its simplicity, the Greedy algorithm is effective when used in the context of the multilevel process. At the coarser levels, the vertices that are moved are actually clusters of vertices in the original graph, allowing for significant changes in the partitioning. At the finer levels, Greedy refinement is able to move vertices that had originally been contracted together into separate partitions.

Further decreasing the edgecut beyond a local minima, requires moving more than one vertex. These groups of vertices whose movement presents a net decrease in the

edgecut are referred to as *hills*. The process of moving these groups of vertices to decrease the edgecut is referred to as *hill-climbing*. The capability to hill-climb, defines a class of high-quality refinement techniques.

One of the earliest methods for refining a two-way partition is that of the Kernighan-Lin algorithm [96]. Originally proposed as a direct means of inducing a partition, it works by first randomly assigning vertices to each partition. It then goes through the vertices and identifies the most beneficial pairs of vertices to swap between partitions. It does this continually until all vertices have been swapped. It then reverts back to the best state of the partition that was observed while performing these swaps. This process repeats until no improved states are found. In its original form this method has an $O(n^2 \log n)$ runtime. It is shown that this bisection (2-way partitioning) method can be used to create k-way partitionings via recursion. This process of recursively splitting a graph to achieve a k-way partitioning is known as *recursive bisection*.

Fiduccia and Mattheyses [97] improved upon this method by relaxing the balance constraint and moving vertices one at a time instead of swapping. Priority queues are used to identify the order in which to move vertices. Again, all possible moves are made, before the algorithm reverts back to the best observed state. For arbitrarily weighted graphs, this algorithm runs in $O(m \log n)$ time ($O(m)$ time for uniform edge weights using a special data structure).

Gong and Lim [112] proposed k-way Pairwise FM (KPM) refinement, which identifies independent pairs of partitions, and performs FM on these pairs. A new set of independent pairs is selected and is refined. This repeats until all partition boundaries have had refinement applied. Unlike using FM for recursive bisection, this directly optimizes the k-way edgecut. However, there are $k(k-1)/2$ possible partition boundaries to refine, which can make this a costly method for large numbers of partitions.

Dutt and Deng [113] proposed the CLIP/CDIP variants of FM for hypergraph partitioning. After all vertices have been inserted into the priority queue, they have their priority all set to zero while preserving the ordering. Then, the top vertex v is extracted and moved, and all of its neighbors have their priorities updated. This has the effect of restricting the search space of FM to the moved vertex v.

Sanders and Schulz [114] introduced a variant of FM which also focuses on localized vertex moves, but in a k-way setting. Their variant, named Multi-Try FM, uses multiple

small trials of FM per iteration. A trial starts by inserting a random vertex into the priority queue, the seed vertex. Once this vertex is extracted and moved, its neighbors are added to the priority queue. One sided FM then continues with the restriction that vertices only move from the same source partition to the same destination partition as the seed vertex for the trial. A new seed vertex is selected, and this process continues until all vertices in the graph have been visited. The complexity of this algorithm is kept to $O(m \log n)$ by marking vertices as visited when the enter the priority queue, and preventing them from re-entering it until the next iteration.

Hill climbing has also effectively been accomplished by re-coarsening a graph and then applying refinement at coarser levels. This approach, although computationally expensive, has been show to result in high quality partitionings [93, 115, 111].

4.1.4 Hypergraph Partitioning

A hypergraph is an extension of the graph model, in which edges may span more than two vertices. That is, a hypergraph H is composed of a set of hyperedges (nets) and a set of vertices. Each hyperedge is an unordered set of one or more vertices. The hypergraph partitioning problem has its origins in VLSI placement [116]. It has since been applied to the efficient storage of databases on disk [117] and directly minimizing the communication volume distributed memory parallel sparse matrix vector multiplication [118].

Compared to graph partitioning, hypergraph partitioning is compute and memory intensive. This is due to two factors. The first is that as a vertices on a hypergraph are contracted, the number of hyperedges tends decrease much slower. On a graph when two vertices are contracted, the edge between them is removed, and any edges with the same endpoint are combined. However, hyperedges may contain more than two vertices decreases the frequency with which these two cases occur. It also becomes more computational intensive to check if two hyperedges have identical vertex lists and should be combined. The second is that refining a partitioning of a hypergraph requires significantly more computations. This is because when a vertex v is moved, all vertices in all hyperedges incident to v to need to have their priorities completely recalculated.

Several hypergraph partitioning packages exist today based on the multilevel paradigm:

hMetis [93], *PaToH* [118], *Zoltan* [119], and *Park*way [120]. Rajamanickam and Boman [121] examined under what conditions directly minimizing communication volume is worth the high cost associated with hypergraph partitioning. The result was that for graphs which are well suited for the summation and bipartite, hypergraph partitioning offer little in the way of quality compared edgecut based graph partitioning. However, for cases where the summation and bipartite graph models do not apply well, and the computation to be performed on the underlying data is significant, hypergraph partitioning can be a valid option.

4.1.5 Vertex Separators

The straight forward method of finding a vertex separator in graph given an edge separator which bisects the vertices in the graph in A and B, is select either the vertices A' in side A which are incident to the cut edges, or the vertices B' in side B which are incident to the cut edges. A more effective method however, is to compute a minimum vertex cover of the bipartite induced by the cut edges [83]. It has been shown that the resulting separator is $10 - 25\%$ smaller of A' and B' [122]. Within the context of the multilevel paradigm, this conversion of an edge separator to a vertex separator can be done at the initial partitioning phase [110] and a modified version [123] of the FM refinement algorithm can be used to reduce the size of the separator as it is projected back down to the finer graphs. This can allow for the finding of small vertex separators that are not co-located with small edge separators.

Hager and Hungerford [124] originally used a bilinear programming formulation to solve the vertex separator problem. While it producing results of high quality, this method is computationally expensive when compared to the existing multilevel methods. This was followed with an improvement by Hager et al. [125] that uses the multilevel paradigm to speed up computations, while retaining similar quality.

4.2 Fill-reducing Ordering

4.2.1 Multiple Minimum Degree

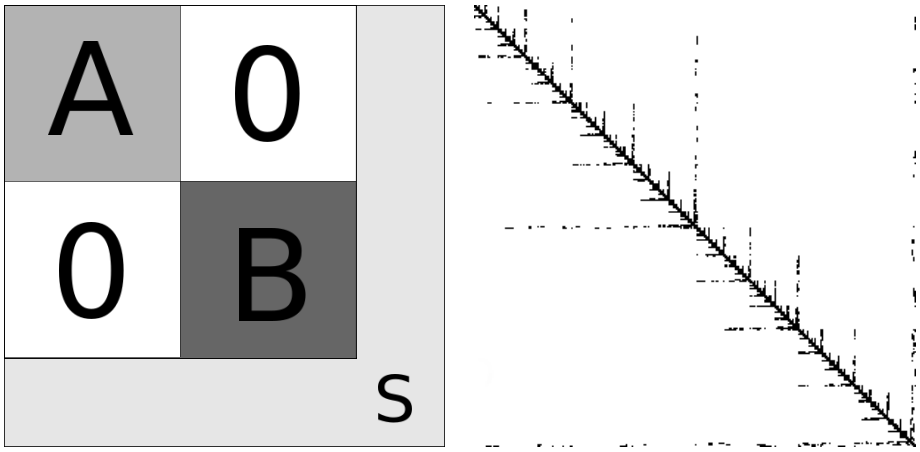
The original minimum degree algorithm proposed by Markowitz [126], works by creating an elimination graph of the matrix, (see Section 2.2 for details on elimination graphs). Vertices are then greedily removed from this graph based on their degree, with the minimum degree being removed first as the name implies. The rows/columns of the matrix are reordered so as to match the order in which the vertices were removed.

Liu [127] proposed the Multiple Minimum Degree algorithm (MMD), which greatly reduces number of degree update operations performed. When there is more than one vertex with the minimum degree in the elimination graph (as is often the case), an independent set of these vertices with minimum degree is selected. This set is the removed simultaneously. This will still result in an ordering obtainable using the Minimum Degree algorithm, as it can be seen that if just one of the vertices from this independent set are eliminated, the remaining vertices will still have the minimum degree among remaining vertices. The gain the associated with removing this set at once is that if two vertices in the set have a common neighbor, it only needs to be updated once. Heggernes et al. [128] analyzed the complexity of the various minimum degree algorithm variants. They show that the MMD algorithm runs in $O(n^2m)$ time.

4.2.2 Nested Dissection

Originally proposed by George [26, 27], nested dissection works by recursively partitioning the graph representation of a symmetric sparse matrix via vertex separators, ordering the rows and columns with partition A first, B second, and S last as shown in Figure 4.3. This new ordering can greatly reduce the required memory and number of computations. Because at each level the vertex separators induce (at least) two connected components, A and B, parallelism can efficiently be extracted by ordering A and B in parallel. These independent components derived during nested dissection also allow for parallelization when computing the Cholesky factorization [129, 130].

The distributed memory parallel algorithm for nested dissection developed by Karypis and Kumar [122] is currently considered the state of the art [131]. This algorithm works by first statically assigning vertices to processors. Then, the processors cooperatively



(a) (b)

Figure 4.3: A reordering of a matrix via a vertex separator 4.3(a) and the non-zero pattern of a sparse matrix reordered recursively via vertex separators 4.3(b).

work to find a vertex separator S, dividing the graph into parts A and B. After the initial separator is found, processors continue to work together finding separators recursively of A and B until p parts have been found, without changing the assignment of vertices. This is done to reduce the volume of communication required to partition the graph down to p parts. Once at least p parts have been found, the graph is moved between processors until each processor has an entire part, and then each processor can compute the nested dissection of their part independently.

4.3 Graph Clustering

A large number of approaches for maximizing modularity have been developed since it was first proposed. Fortunato [132] provides an overview of modularity and methods for its maximization.

Agglomerative Methods

The majority of approaches fall into the category of agglomerative clustering. In agglomerative clustering, each vertex is placed in its own cluster, and pairs of clusters are

iteratively merged together if it increases the modularity of the clustering. When there exists no pair of clusters whose merging would result in an increase in modularity, the process stops, and the clustering is returned.

The greedy agglomerative method introduced by Clauset et al. [133], is the most well-known of the these approaches, due to its ability to find good clusterings in relatively little time. Its low runtime is the result of exploiting the sparse structure of the graph to limit the number of merges it needs to consider and the number of updates that it needs to perform during agglomeration. The quality of the clusterings it finds is the result of recording the modularity after each merge, and continuing to perform cluster merges until there is only a single cluster, and then reverting to the state with the maximum modularity. The structure used to maintain this state information is a binary tree in which each node represents a cluster, and the children of a node are the clusters which were merged to form the node. They established an upper bound on the complexity of this algorithm of $O(mh \log n)$, where h is the height of the tree recording cluster merges. If this tree is fairly balanced, h will be close to $\log n$.

It was noted that this algorithm tends to discover several super-clusters, composed of most of the vertices in the graph. Wakita and Tsurumi [134] showed that these super clusters are the result of one or a few large clusters successively merging with small clusters, causing h to approach n, which results in a running time near $O(mn)$. They also showed that the creation of these super-clusters can be of detriment to the modularity of the clustering. They addressed this by presenting an algorithm that favors merging clusters of similar size, which helps to prevent this unbalanced merging.

Although it does not maximize modularity explicitly, Label Propagation [135] is an iterative scheme that starts by assigning every vertex a unique label, and in every iteration a new label is assigned to each vertex based on the label of the majority of its neighbors. Although it does not maximize modularity as well many of the agglomerative schemes, its near linear running time still makes it an attractive option for maximizing modularity on large graphs.

The Louvain method [136] finds a set of cluster merges through a recursive process. It does this by initializing every vertex to its own cluster as is done in agglomerative methods, and then for each vertex, checks to see if moving it to a different cluster will improve modularity. It moves vertices this way in passes, until a pass results in no

moves being made. Then, a new graph is generated where each vertex is a cluster of vertices from the previous graph. This process is repeated recursively until a graph is generated in which no vertices change clusters. This is currently one of the fastest modularity based clustering methods available [137].

There is a small number of parallel algorithms for modularity based graph clustering. Reidy et al. [138] generate new graphs similar to the Louvain method. However, here instead of moving vertices, clusters are merged by collapsing a maximal matching of the clusters. Parallelism is extracted by calculating the desirability to collapse each edge independently, and then a multi-pass method is used to find the maximal cluster matching. Fagginger Auer and Bisseling [139] present a similar approach using maximal matchings on GPU architectures, with extensions to matching in order to increase the rate of cluster merging. Both of these use a fine grain approach to parallelism, and are similar to the coarsening phase of the multilevel paradigm discussed in the next section.

Staudt and Meyerhenke [140] developed a parallel version of the Label Propagation algorithm. Their algorithm takes advantage of the independent nature of determining the label for each vertex, and as a result scales quite well. However, as is the case with the serial formulation of label propagation, it does not directly optimize modularity and can fail to produce clusterings with high modularity. Along with parallel label propagation, Staudt and Meyerhenke also proposed a parallel version of the Louvain method, which visits vertices in parallel and moves them between clusters using possibly stale cluster information. To further improve the quality of these clusterings, they also added a secondary move step (referred to as refinement) after the Louvain method has been recursively applied.

Multilevel Methods

Noack and Rotta [141] developed a method for modularity based graph clustering that uses the multilevel paradigm. Instead of collapsing independent sets of vertices as in graph partitioning, they use agglomerative clustering to iteratively determine groups of vertices to collapse together. To avoid the uneven merging of clusters, they prioritize cluster merges based on $\Delta Q / \sqrt{d(C_i)d(C_j)}$, where ΔQ is the gain in modularity from merging clusters C_i and C_j. The state of the clustering is intermittently instantiated as a graph to provide several levels on which refinement can be performed. Their refinement

visits each vertex and considers it for moving between clusters.

Djidjev and Onus [142] showed that the multilevel algorithms of [92] for graph partitioning can be used directly to find two-way clusterings with high modularity by using a modularity derived input graph.

Chapter 5

Shared Memory Multilevel Graph Partitioning

In this chapter we explore the design space of creating a multithreaded graph partitioner. We present and compare various algorithms for the key steps of the multilevel partitioning paradigm that take different strategies in terms of synchronization frequency, task granularity, data ownership, and thread lifetime. Our experiments, on three different multicore architectures using the threading functionality provided by OpenMP, show that even though multicore architectures allow for fine grain task decomposition and frequent synchronization, the best performance is achieved when the task decomposition is coarse and synchronization is infrequent. In addition to this we found that data locality, which when using OpenMP can only be controlled implicitly by having threads operate on the data they generate, is also crucial to achieving performance on a large number of cores. These findings are to a large extent consistent with the best practices of high performance parallel algorithms used on distributed memory machines. We present carefully designed coarsening, initial partitioning, and uncoarsening techniques that result in greater than a factor two speedup over other parallel partitioners. In addition, without the need to cache remote data locally on multicore architectures, we were able to significantly reduce the aggregate amount of memory required as the number of threads increases compared to distributed memory formulations.

5.1 Methods

5.1.1 Coarsening

The coarsening phase of multilevel graph partitioning first computes a vertex matching and then builds the next-level coarser graph via graph contraction in which the matched vertices are combined and the adjacency lists of the combined vertices are combined.

As discussed in Section 4.1.3, the Heavy Edge Matching algorithm visits the vertices of the graph in ascending order of degree. For each unmatched vertex, it matches it with the adjacent unmatched vertex that is connected via the highest weight edge. If no such vertex exists, then the vertex remains unmatched. In parallelizing this algorithm, we followed an approach in which the vertices of the graph are divided among the different threads, and each thread is responsible for matching the vertices assigned to it.

A direct implementation of this approach will use a shared matching vector M. Each thread will read this vector in order to determine the matching status of vertices, and write to it every time it matches a pair of vertices together. In order to ensure that there are no race conditions, each read/write operation on the shared vector M will need to be protected via a lock. This will result in excessive locking and lead to poor parallel performance. An alternate approach is for each thread to read M without locking but ensure that the write operations (i.e., the matching decisions) are valid. This is achieved as follows. Once a thread has chosen to match vertex v with vertex u, it locks both $M(v)$ and $M(u)$ and performs a final check to make sure that both vertices are still unmatched. If this holds, it then sets $M(v) = u$ and $M(u) = v$ before releasing the locks on them. Locks are acquired in ascending order to prevent deadlocking. We will refer to this scheme as *fine-grain matching*.

Even though the above approach reduces the amount of time spent in locks/unlocks over the naive approach, we expect that it will still incur substantial shared data access synchronization overheads. For this reason, we evaluated another approach that is similar to the iterative two-pass approach used by Karypis and Kumar [143]. Specifically, the vertices of the graph are initially divided among the threads. Each thread matches its vertices by giving preference to unmatched adjacent vertices that are also assigned to the same thread. Each thread also has a *request buffer* for each other thread. The matchings that involve adjacent vertices assigned to other threads are placed into the *request*

buffer corresponding to that thread. During the second pass, each thread processes the corresponding request buffers of other threads and either accepts or rejects the requested matches for its vertices and modifies M accordingly. After several passes, any unmatched vertices are matched with themselves. We will refer to this scheme as *multi-pass matching*.

The multi-pass matching approach addresses the high synchronization overheads of the fine-grain matching approach but it introduces the extra cost of maintaining and servicing the request buffers. The third approach relies on the heuristic nature of matching and exploits ideas from both the fine-grain and the multi-pass matching approaches. Just as in the fine-grain approach each thread writes to M for both local and non-local vertices. However, unlike the fine-grain approach, the writes in M are done without locking. This makes it possible for a vertex v to believe that it is matched to the vertex u, while u believes it is matched to the vertex w (i.e., $M(v) = u$ and $M(u) = w$). To correct this, after M is generated, each thread goes through its list of local vertices and any vertex v in an asymmetrical matching $M(v) = u$ and $M(u) \neq v$, are matched with themselves. As long as the number of vertices in the graph is much greater than the number of threads, these asymmetrical matchings occur infrequently so as not to disturb the size of the matching. In our experiments we observed 0.001% of vertices involved in asymmetrical matchings at the finest level, and 0.13% at the coarsest level. This corresponded to about $120,000$ vertices per thread at the finest level, and about $1,000$ vertices per thread at the coarsest level. This unprotected approach attempts to gain the best of both approaches, by avoiding the synchronization overheads of the fine grained approach, and the extra memory accesses required for handling requests in the multi-pass approach. This is similar to an approach explored by Çatalyürek et al. [144] for aggregating hypergraphs. We will refer to this scheme as *unprotected matching*.

Once we have populated the matching vector M, a parallel prefix-sum is performed over the matchings, so that in a second pass coarse vertex numbers can be assigned to each match in parallel, generating C.

With the matching vector M and its associated vertex mapping vector C, the parallelization of the graph contraction operation on a shared memory system is straightforward. Irrespective of the scheme used to compute the matching, our parallel graph contraction algorithm operates as follows. The vertices of the next-level coarse graph

are divided among the threads and each thread is responsible for merging the adjacent lists of the corresponding matched vertices.

5.1.2 Initial Partitioning

An initial partitioning is created by using recursive bisection to generate the desired k-way partitioning (see Chapter 3 for details). Since initial partitioning will always be performed on a small problem size, the design space in which it can effectively be parallelized is quite small. The two approaches we explore are parallelizing each bisection, *parallel bisectioning*, and parallelizing all of the k-way partitionings, *parallel k-sectioning*.

In parallel bisectioning, each thread bisects G into A and B, and the best bisection is selected. The threads are then split into two groups, and one group recursively performs a parallel bisectioning on A and the other recursively performs a parallel bisectioning on B. This is done until a k-way partitioning is obtained. This requires the threads to synchronize $\log_2 k$ times to perform a min-reduce operation and select the best partitioning. At each bisection, 16 partitionings are made to both ensure a quality bisection is selected and to provide enough useful computation to parallelize.

In parallel k-sectioning, each thread independently generates k-way partitionings of G_m via recursive bisection, and the best partitioning among all of the threads is selected. Among all of the threads, 16 k-way partitionings are generated to ensure a quality initial k-way partitioning can be selected as well as to provide enough useful computation to parallelize. The only synchronization point is the min-reduce operation at the end to select the best partitioning.

5.1.3 Uncoarsening

The uncoarsening phase of the multilevel graph partitioning paradigm consists of two steps. First, the partition labels P_{i+1} of the next-level coarse graph are projected to the current coarse graph in order to compute P_i, and then a greedy move-based refinement algorithm is used to further reduce the edgecut of the resulting k-way partitioning.

Since all data is accessible by all threads, including the partition label vector P_{i+1}, the projection step can be easily parallelized by dividing the vertices among the threads,

and having each thread determine the partition label of its assigned vertices. Memory bandwidth and latency can become limiting factors when attempting to achieve high speedup with a large number of cores in this step.

Parallelizing refinement is considerably more complicated than projection. First, in order to ensure concurrent refinement the global greedy strategy needs to be relaxed. Second, as different threads move vertices among partitions concurrently, care must be taken to ensure that balance is maintained. For example, the partitioning solution can become unbalanced if thread a considers moving vertex v to partition V_i, and at the same time thread b considers moving vertex u to partition V_i. When both threads check to make sure their moves will result in balanced partitions, they see $|V_i| + \eta(v)$ and $|V_i| + \eta(u)$ as being below the maximum allowable partition weight, but the resulting weight of $|V_i| + \eta(v) + \eta(u)$ is greater than the maximum allowable partition weight. Third, the concurrent move of vertices can also lead to a degradation of the edgecut, even if these moves were initially selected based on greedy strategy. This happens in the cases in which the vertices that are moved concurrently are connected via an edge. For example, consider two adjacent vertices v and u belonging to partitions V_i and V_j respectively, with $\theta(\{v, u\})$ being greater than the sum of the weights of the rest of the edges incident on v and u. In a situation like that, moving either v or u to the other vertex's partition, will improve the edgecut. However, if these two vertices were assigned to different threads, and each thread decided to perform the move, then the overall edgecut may increase rather than decrease. Fourth, the serial implementation of the Greedy refinement algorithm considers only the vertices that are at the partition boundaries and pre-computes the per-adjacent partition cuts for each boundary vertex. This information is efficiently updated as vertices get moved, resulting in an extremely fast k-way refinement implementation. As a result, the total time spent in uncoarsening is much smaller than the time spent in coarsening, making the parallelization overheads significant compared to the useful computation performed.

To address the above challenges, we developed two different parallel formulations of the Greedy refinement algorithm. In both approaches, we replaced the single global priority queue with per-thread priority queues. Thus, the boundary vertices are divided among the threads, each thread inserts them in its own priority queue based on their gain, and then proceeds to process them. Note that even though an approach like

that may not be as effective as a globally greedy strategy, our experience with schemes like that in *ParMetis* has shown that their overall impact on partitioning quality is minimal. However, the two formulations differ on how they ensure balance, how they deal with potential cut degrading moves, and how they update the refinement-related data-structures.

The first approach attempts to keep the partition information up to date by performing updates as each move is made. Once a thread has removed a vertex v from the top of its priority queue, it finds the best eligible partition to move v to. A partition V_i is eligible only if its weight plus the weight of v is under the maximum allowable partition weight. The best partition is the one to which the sum of the weight of the edges connecting v to the partition is the greatest. If the best partition for v is the one it is already in, or no eligible partition can be found, v is not moved. Otherwise v and all of its neighbors $\Gamma(v)$ are locked using the same hashing technique as in Section 5.1.1, and both the partition containing v and partition receiving v are locked as well. Final checks are then performed to ensure the pending weight changes will be within the balance constraint, and that the move will still result in an edgecut reduction. Locking the partition weights and performing the final check, ensures the balance of the partitioning is preserved. Locking v and its neighbors ensures that no moves that increase the edgecut are made, and that the refinement-related data-structures are consistent. Finally, once the updates have been performed, the locks are released. We refer to this approach as *fine-grain refinement*.

There are three potential limitations with the above approach. First, it will tend to incur a high synchronization overhead due to the frequent locking. Second, it requires the number of partitions be greater than the number of threads in order to keep the updating of the partition weight from becoming a bottleneck. Third, since the precomputed refinement-related data-structures are updated by multiple threads, it can lead to false sharing.

To address these issues, the second approach attempts to minimize direct interaction between threads and utilizes message passing. To ensure that two vertices connected by a sufficiently heavy edge will not swap partitions and increase the edgecut, vertices are restricted to moving across partition boundaries in only one direction at a time. When a thread removes a vertex v from its priority queue, it decides whether or not to move

v following the same process as fine-grain refinement (described above), with the added restriction of move direction.

Additionally, each thread has an *update buffer* for each other thread. If it decides to move a vertex v, it updates its local vertices and places updates to adjacent non-local vertices into its corresponding update buffer. After each thread has removed a fixed number of vertices from its priority queue, all threads communicate what the potential partition weights would be after their moves are committed. The balance of the k-way partitioning is maintained by undoing pending moves until the remaining moves would result in a balanced partitioning. The remaining moves are then committed and threads update their local vertices' partition connectivity for the moves made by other threads. This is then repeated until all of the priority queues are emptied. We refer to this approach as *coarse-grain refinement*.

For both approaches, a pass ends when the priority queues of all threads are empty. Refinement terminates when the maximum number of passes has been reached, or no vertices were moved in the last refinement pass.

5.1.4 Thread Lifetimes

Our discussion so far has focused on algorithmic aspects related to the frequency of synchronization and task granularity. However, another equally important aspect has to do with when and for how long spawned threads live, which we will refer to as *thread lifetime*. In this work we explore two different approaches, which we will refer to as *fork-join* and *thread-persistence*.

In the fork-join approach, threads are started at the beginning of a parallel block of work and stopped at the end. This is the paradigm around which OpenMP was created. In the multilevel graph partitioning algorithms we have discussed so far, this means starting threads at the start of matching and joining them at the end, and doing the same for each of contraction, initial partitioning, projection, and refinement. Note that whether threads are actually spawned and killed for each parallel block of work or pulled from an existing thread pool is implementation dependent.

The thread-persistence approach creates all of the threads at the start of program execution, similar to MPI, and does not join them again until the program's termination. This approach results in some work duplication between threads, but results in fewer

synchronization points.

5.1.5 Data Ownership

A design space tightly coupled with thread lifetimes is that of data ownership. In this context, data ownership refers to a thread performing work on a fixed set of vertices and their incident edges. We investigated three approaches to data ownership: one which assigns work dynamically, one which assigns works statically at the start of each block of parallel work, and one for which data ownership persists throughout the entire execution the multilevel paradigm. The fork-join model of OpenMP restricts data ownership to within a parallel block. As a result, for the third approach this is accomplished while using the thread-persistence described above.

The first approach has no concept of data ownership and uses dynamic work scheduling. This provides the benefit of dynamic load-balancing at the cost of increased overhead for distributing the work. This approach also fails to preserve data locality. For the multilevel paradigm, this means that threads pull vertices from a shared pool on which to perform matching, contraction, projection, and refinement as described in Sections 5.1.1 and 5.1.3. The initial partitioning approaches we took in 5.1.2 are unaffected by this design space as each thread gets its own copy of the coarsest graph. We will refer to this approach as *dynamic work-distribution*.

In a second approach to data ownership, threads are given ownership of data statically at the start of a block of parallel work. This approach relies on the even distribution of work at the start of each parallel work block, but does not suffer from the overhead associated with dynamic load-balancing. Data locality with this approach is limited to the parallel work block. In the multilevel graph partitioning paradigm, this means vertices are divided among the threads at the start of matching, contraction, projection and refinement for each of the graphs $G_0, G_1, \ldots, G_s$. We will refer to this approach as *static work-distribution*.

In the third approach, threads are given ownership of data at the start of the program, and continue to own that data and all derived data for the duration of the program. This approach is not possible with the fork-join approach to thread lifetimes. This approach ensures the locality of the data worked on by each thread. However, not only does this approach lack dynamic load-balancing, it also provides no guarantee that

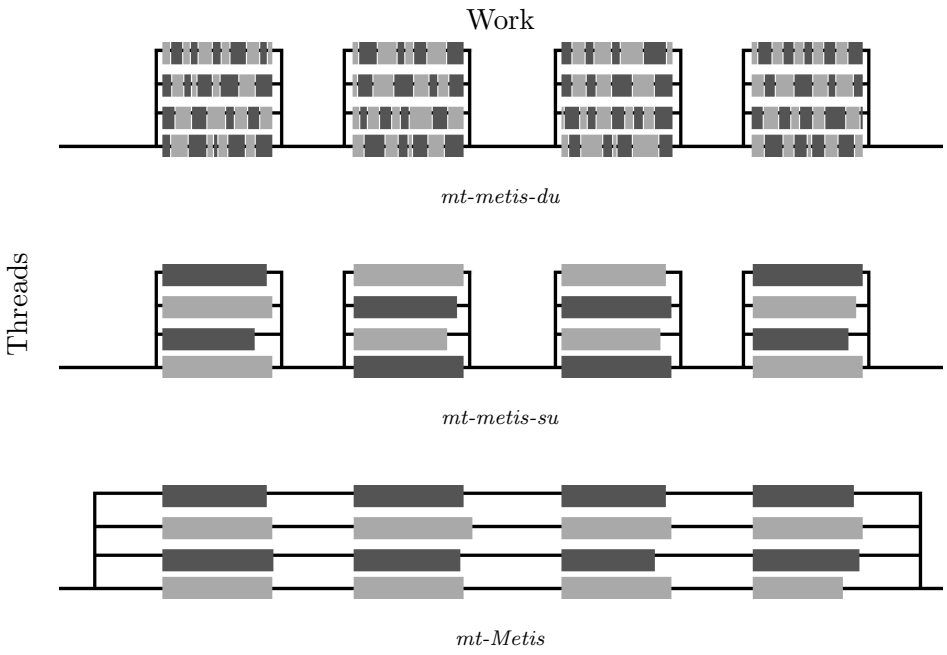


Figure 5.1: The threading and work distribution models for *mt-metis-du*, *mt-metis-su*, and *mt-Metis*, where a line represents a thread and the gray blocks represent chunks of work.

the work will be evenly distributed at the start of a work block. In the context of the multilevel graph partitioning paradigm, this means that the vertices of G_0 are initially divided among the threads, and in G_1 threads own the coarse vertices they created while contracting the fine vertices of G_0 that they owned. We will refer to this approach as *persistent work-distribution*.

For evaluating these data ownership approaches as well as the associated thread lifetime approaches discussed in Section 5.1.4, we developed three OpenMP based implementations of the algorithms discussed in Sections 5.1.1, 5.1.2, and 5.1.3. The first, *mt-metis-du*, uses the fork-join approach for thread lifetimes and the dynamic work-distribution approach for data ownership. The second implementation, *mt-metis-su*, also uses the fork-join approach for thread lifetimes, but uses the static work-distribution approach for data ownership. The third implementation, *mt-Metis*, uses the thread-persistent approach to thread lifetimes and the persistent work-distribution approach for data ownership. These differences are illustrated by Figure 5.1.

5.2 Experimental Design

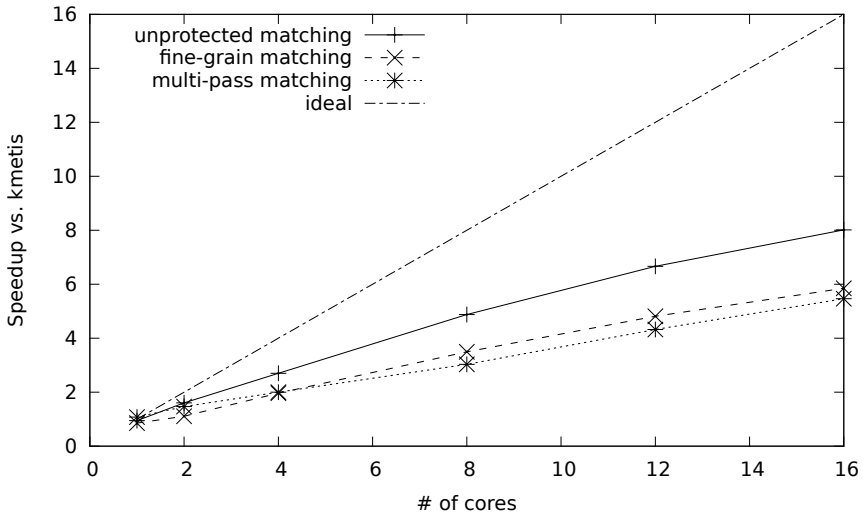
Each run was repeated 50 times with a different random seed, and the average time and edgecut were taken. An imbalance tolerance of 3% was used for all partitionings, all of which were 64-way. Run times are only for the three phases of multilevel partitioning, and not for IO. We used the k-way partitioning portion (referred to as *Metis*) of *Metis* 5.0.2, *ParMetis* 4.0.2, and *Scotch* 5.1.2. The default settings were used for both *Metis* and *ParMetis*. Both the *scalability* and *speed* flags were used when running *Scotch*.

Speedup is measured with respect to the runtime of *Metis*. Where speedups are aggregated for all four graphs, the geometric mean has been taken of their speedup with respect to *Metis*. Edgecut is measured relative to *Metis*. Where edgecuts are aggregated for all four graphs, their geometric mean has been taken relative to *Metis*.

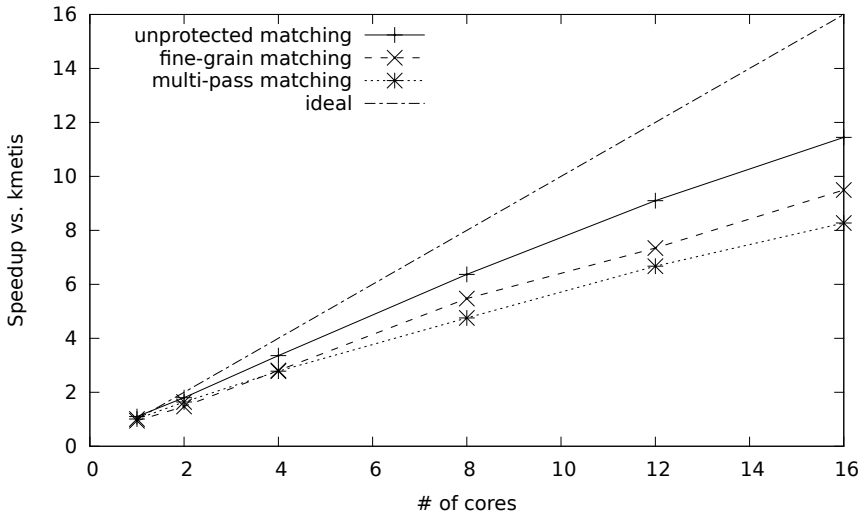
Three systems were used for these experiments: an HP ProLiant BL280c G6 with 2x 8-core Xeon E5-2670 @ 2.6 GHz, a Dell PowerEdge R815 with 4x 8-core Opteron 6220 @ 3.0 GHz, and a Sun Fire X4600 with 8x 4-core Opteron 8356 @ 2.3 GHz. Codes were compiled using Intel's ICC compiler version 11.1 on the 8x 4-core Opteron system, and 11.2 on the 2x 8-core Xeon and 4x 8-core Opteron systems, all with *O3* optimizations enabled. Regarding the thread lifetimes discussed in Section 5.1.4, Intel's implementation of OpenMP, as used in these experiments, makes use of thread pooling [145]. We were not able to produce reliable results when utilizing all 32 cores of the 4x 8-core Opteron system for any of the partitioners, and as a result we only report using up to 28 threads/processes on this system.

5.3 Results

Our experimental evaluation consists of three parts. First we evaluate the impact of the various algorithmic choices for the different phases of the multilevel paradigm as it relates to task granularity and synchronization frequency. Second, we evaluate the impact of thread lifetime and data ownership. Finally, we evaluate the performance of the best performing algorithms in the multithreaded graph partitioner *mt-Metis* against the performance achieved by two MPI-based parallel multilevel algorithms, *ParMetis* and *Scotch*.



(a) mdual2

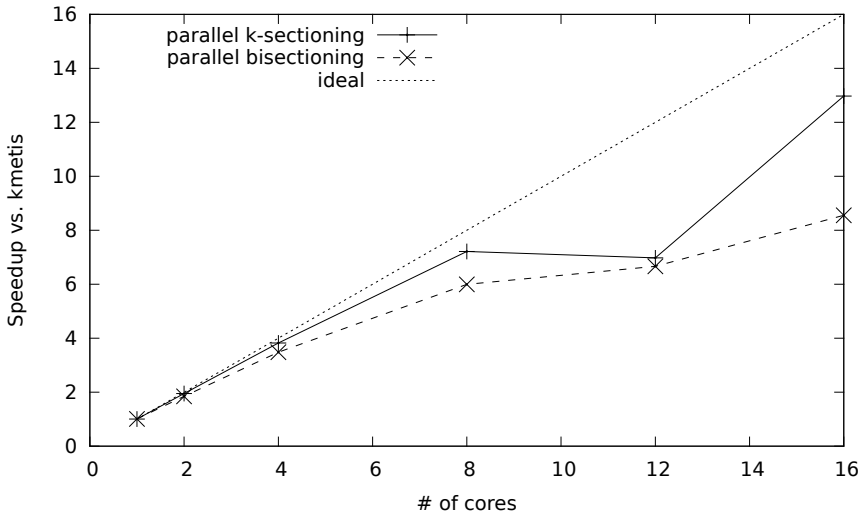


(b) vlsi_crct

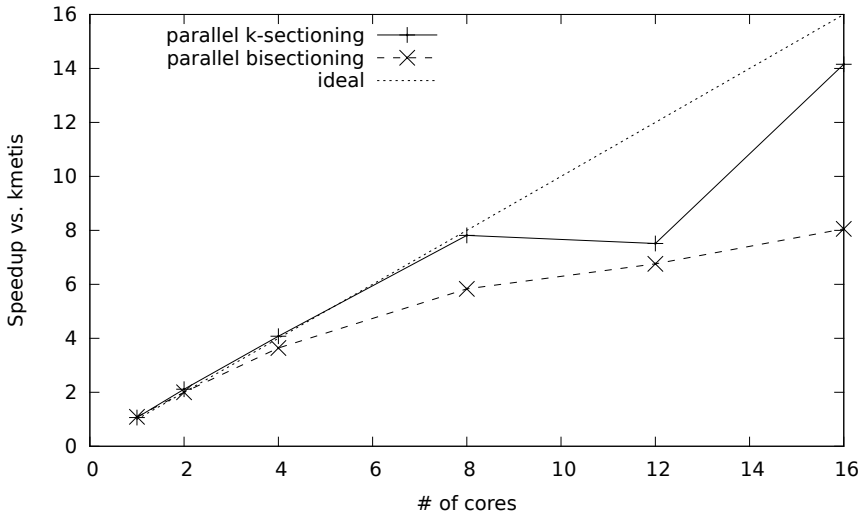
Figure 5.2: Coarsening

5.3.1 Granularity of Parallelism

To study the effect of the various algorithmic choices for coarsening, initial partitioning, and refinement, we performed a series of experiments in which the algorithms were implemented and evaluated using the *mt-metis-du* framework as described in Section



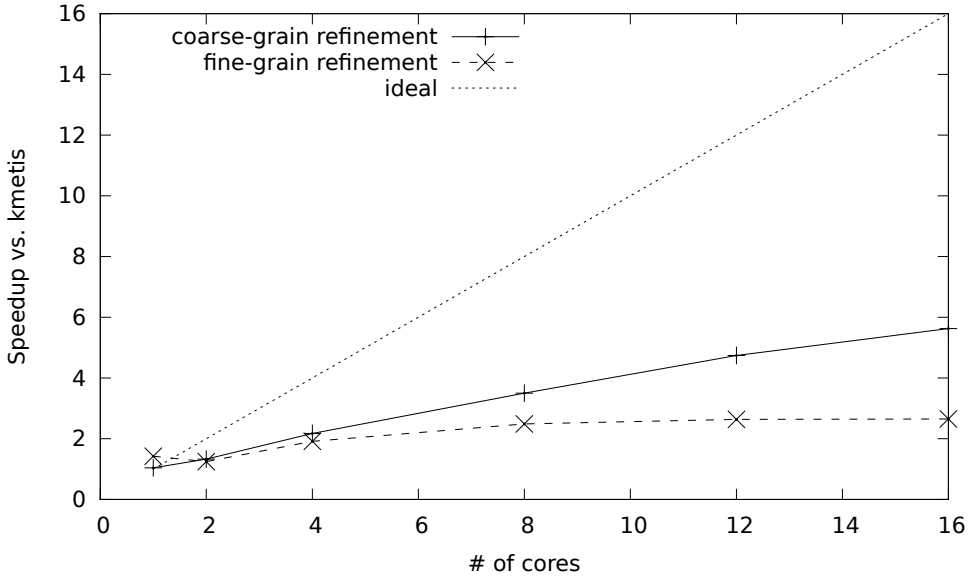
(a) mdual2



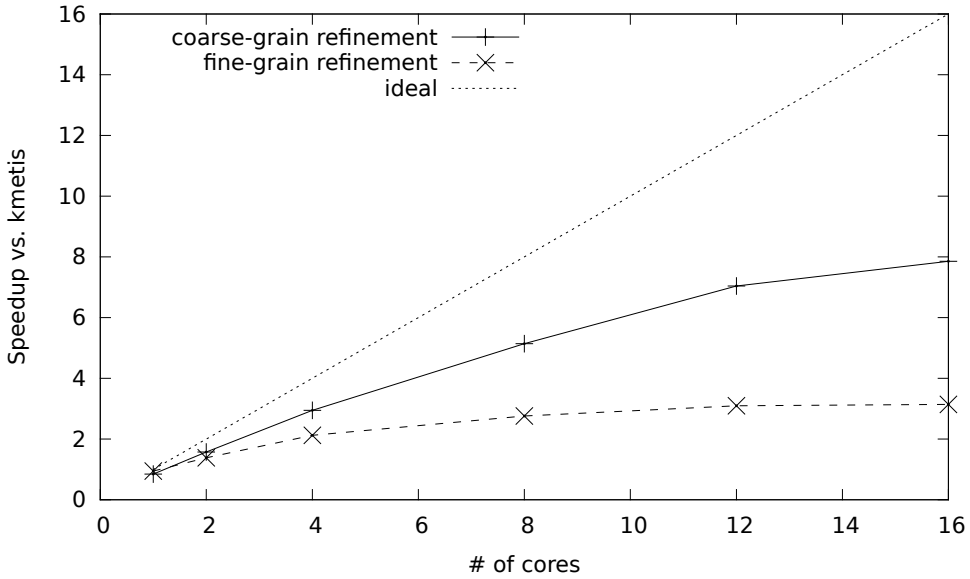
(b) vlsi_crct

Figure 5.3: Initial Partitioning

5.1.5. We present results for the smallest and largest graphs in our dataset, `mdual2` and `vlsi_crct`, on the 2x 8-core Xeon system.



(a) mdual2



(b) vlsi_crct

Figure 5.4: Uncoarsening

Coarsening

The results of the three coarsening approaches: fine-grain matching, multi-pass matching, and unprotected matching are shown in Figure 5.2. The results show that the unprotected approach outperformed the other two. The better performance of the fine-grain approach over the multi-pass approach can be attributed to the following to reasons. First, the fine-grain approach did not suffer from lock contention because the number of locks was much greater than the number of threads. Second, the overhead associated with acquiring and releasing a lock per matching in the fine-grain approach was slightly less than that of the extra memory accesses required by the request buffer used by the multi-pass approach. The unprotected approach avoids these overheads and using 16 threads achieved a speedup of 8.7 for `mdual2` and 11.4 for `vlsi_crct`, compared to the speedups of 5.5 and 8.3 achieved by the multi-pass approach and the speedups of 5.7 and 9.4 achieved by the fine-grain approach respectively.

Initial Partitioning

The results of the two approaches for initial partitioning, parallel bisectioning and parallel k-sectioning, are shown in Figure 5.3. The parallel k-sectioning approached scaled near linearly when the number of threads was a power of two. This is a result of the 16 partitionings being evenly divided and each partitioning being generated independently. It suffered a slight decrease in speed when using twelve threads because four of the threads still generated two partitionings causing the other eight threads that each only generated one partitioning to wait. Parallel recursive bisectioning did not scale as well because of the increased number of synchronization points. However, because of the finer grain task decomposition, parallel recursive bisectioning still had a balanced work distribution when using twelve threads.

Uncoarsening

The results of the two approaches for uncoarsening, fine-grain refinement and coarse-grain refinement, are shown in Figure 5.4. From these results we can see that the coarse-grain approach outperforms the fine-grain approach. The fine-grain approach manages to gain a small speedup of 2.5 for `mdual2` and 2.8 for `vlsi_crct` using eight

threads, and moves up to a speedup of 2.6 for `mdual2` and 3.1 for `vlsi_crct` using 16 threads. This plateau is likely the result of lock contention for modifying the partition weights. The coarse-grain approach however, steadily gained speedup as the number of threads increased, with a speedup of 3.5 for `mdual2` and 5.1 for `vlsi_crct` using eight threads, and a speedup of 5.6 for `mdual2` and 7.8 for `vlsi_crct` using 16 threads. The greater speedup for the large graph compared to the small graph is because the majority of the work in the uncoarsening phase is only on border vertices, which account for a small fraction of the total vertices in the graph, and as a result the parallel overheads are not as well hidden while performing refinement on the smaller graph.

5.3.2 Thread Lifetimes and Data Ownership

The best granularity strategies for coarsening, initial partitioning, and uncoarsening as determined by the outcome of these experiments, were used in the multithreaded implementations to explore the thread lifetime and data ownership design space discussed in Sections 5.1.4 and 5.1.5. That is, for matching we used the unprotected matching approach described in Section 5.1.1, for initial partitioning we used the parallel k-sectioning method described in Section 5.1.2, and for uncoarsening we used the coarse-grain refinement described in Section 5.1.3.

The mean speedup across all four graphs on each of the three systems for these three implementations can be seen in Figure 5.5. On the 2x 8-core Xeon system we can see that the three multithreaded implementations performed relatively similarly, with *mt-Metis* taking a slight lead when using 16 threads. On the two systems with higher core counts, *mt-Metis* has a prominent lead over *mt-metis-du* and *mt-metis-su* when using more than 16 threads. This performance gap exists because *mt-metis-du* and *mt-metis-su* are not designed to preserve data ownership across synchronization points (i.e., transitioning from matching to contraction, projection to refinement, and moving between levels in coarsening and uncoarsening). When one of these synchronization points are encountered, a thread may work on a different part of the graph than it had previously. Data ownership is preserved by *mt-Metis*, so when the active level of the graph is small enough such that the portion assigned to a thread fits within its cache, it performs the majority of its memory accesses from cache where *mt-metis-du* and *mt-metis-su* are still accessing the slower DRAM the majority of the time. As the

number of threads increases, the portion of the graph assigned to each thread decreases and a larger level of the graph can fit within the cache. This increased data locality enables *mt-Metis* to outperform *mt-metis-du* and *mt-metis-su* in both coarsening and uncoarsening for a large number of threads.

The better utilization of the aggregate available cache also explains the relative performance of *mt-Metis* over the other methods on the three different architectures. Specifically, *mt-Metis* achieved the best relative performance on the 4x 8-core Opteron system that has 1MB of L2 cache/core, its second best performance on the 8x 4-core Opteron system that has 512KB of L2 cache/core, and its worst relative performance (though still better) on the 2x 8-core Xeon system that has only 256KB of L2 cache/core.

5.3.3 Comparison with Other Partitioners

We have compared our best multithreaded implementation, *mt-Metis*, with two publicly available distributed memory partitioners, *ParMetis* [143] and *Scotch* [146].

Speedup

The mean speedup for partitioning the four graphs can be seen for each system in Figure 5.6. In all three of the figures, it can be seen that *mt-Metis* averaged over twice the speedup of both *ParMetis* and *Scotch* when using more than four cores on each of the three systems.

The speedups achieved for partitioning the individual graphs on each system are shown in Table 5.1, and the run times are shown in Table 5.2. The achieved speedups are largely dependent on the graph, and to a lesser extent, the system. The worst that *mt-Metis* performed relative to the other partitioners, was partitioning `mdual2` on the 4x 8-core Opteron system. *ParMetis* reached a speedup of 6.9 compared to the 8.9 speedup of *mt-Metis*. On the same system however, *mt-Metis* saw its largest lead in performance with a speedup of 17.9 compared to *ParMetis*'s speedup of 2.9 and *Scotch*'s speedup of 1.5 when partitioning `vlsi_crct`. As *mt-Metis* and *ParMetis* implement similar algorithms, a great deal of the difference in their runtimes can be attributed to the overheads of message passing. Furthermore, as coarsening is the most time consuming stage of multilevel graph partitioning, *mt-Metis*'s unprotected matching provides a significant advantage over the request based scheme of *ParMetis*.

Table 5.1: Speedup on Individual Graphs (vs *Metis*).

	2x 8-core Xeon		
Graph	*ParMetis*	*Scotch*	*mt-Metis*
mdual2	3.40	0.68	6.56
hd2_fe	2.47	0.37	7.57
road_usa	3.70	1.50	11.66
vlsi_crct	3.91	1.49	12.76
	4x 8-core Opteron (28 cores)		
mdual2	5.20	0.86	7.89
hd2_fe	3.27	0.48	10.54
road_usa	5.68	2.08	14.89
vlsi_crct	4.63	1.41	15.90
	8x 4-core Opteron		
mdual2	6.86	1.31	8.92
hd2_fe	3.14	0.47	12.04
road_usa	4.69	2.20	14.73
vlsi_crct	2.88	1.53	17.89

Note that *Scotch* uses recursive bisection to generate a k-way partitioning [146], causing it to go through the multilevel process several times. On the other hand *ParMetis* and *mt-Metis* only use recursive bisectioning to generate the initial partition, and as a result only go through the multilevel process once. This contributed to its higher runtime seen in our experiments.

Memory Usage

In Table 5.3, we present the aggregate memory usage of these partitioners for creating k-way partitioning of mdual2. Memory usage was measured using the GNU *time* utility. It can be seen that all three of the parallel partitioners increase their memory usage with the number of cores utilized. However, where both *ParMetis* and *Scotch* use over eight times the amount of memory of *Metis* on 32 cores, *mt-Metis* only uses 44% more than *Metis*, and only 13% more than it did running serially. Thread private data structures

Table 5.2: Time on Individual Graphs (seconds).

	2x 8-core Xeon		
Graph	*ParMetis*	*Scotch*	*mt-Metis*
mdual2	0.233	1.168	0.121
hd2_fe	1.421	9.566	0.463
road_usa	7.401	18.276	2.351
vlsi_crct	17.416	45.653	5.337
	4x 8-core Opteron (28 cores)		
mdual2	0.281	1.697	0.185
hd2_fe	1.892	12.921	0.588
road_usa	6.884	18.817	2.627
vlsi_crct	19.387	63.671	5.651
	8x 4-core Opteron		
mdual2	0.585	3.057	0.450
hd2_fe	6.746	45.146	1.761
road_usa	24.445	52.155	7.779
vlsi_crct	96.578	181.391	15.553

used during coarsening and uncoarsening account for the slight increase in memory usage by *mt-Metis* as the number of threads increases. The large difference in memory usage between *mt-Metis* and the MPI based partitioners is in large part because *mt-Metis* stores information for each vertex only once, where *Scotch* and *ParMetis* need to communicate and store the information of remote neighbor vertices.

Partition Quality

To ensure a valid comparison, we studied the average and minimum number of cut edges of the partitions of the four graphs generated by the partitioners. Each partitioner generated 50 partitionings of each graph. The geometric mean of both the average edgecut and minimum edgecut relative to *Metis* are shown in Tables 5.4 and 5.5. The tables show that the quality of partitionings created by *mt-Metis* does not diverge from that of *ParMetis* and *Scotch*.

Table 5.3: Memory usage (MB).

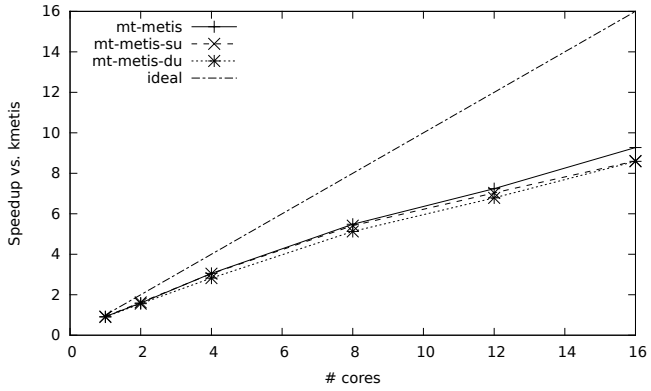
	Cores					
Name	1	2	4	8	16	32
Metis	522	-	-	-	-	-
ParMetis	522	1,218	1,501	1,895	2,899	4,691
Scotch	593	742	1,019	1,241	2,251	5,001
mt-Metis	665	680	700	696	742	752

Table 5.4: Geometric means of average cuts scaled relative to *Metis*.

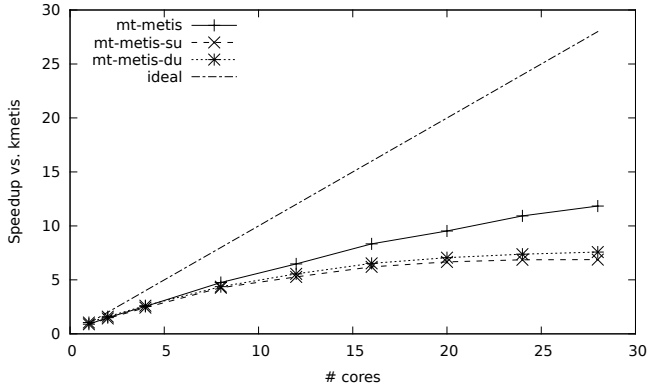
	Cores					
Name	1	2	4	8	16	32
Metis	1.000	-	-	-	-	-
ParMetis	1.000	1.131	1.221	1.116	1.113	1.108
Scotch	1.111	1.113	1.113	1.089	1.102	1.100
mt-Metis	1.075	1.072	1.076	1.085	1.104	1.102

Table 5.5: Geometric means of minimum cuts scaled relative to *Metis*.

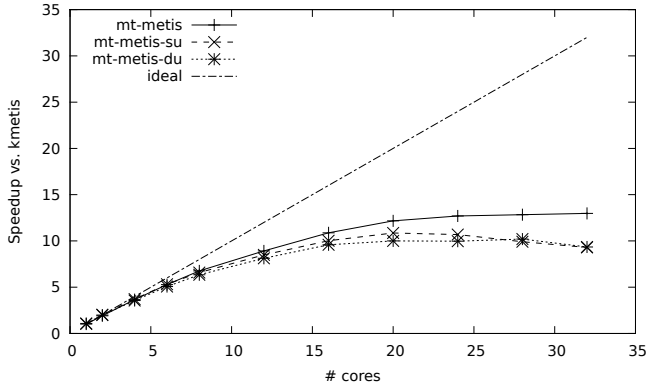
	Cores					
Name	1	2	4	8	16	32
Metis	1.000	-	-	-	-	-
ParMetis	1.000	1.063	1.060	1.056	1.049	1.047
Scotch	1.037	1.039	1.037	1.042	1.029	1.033
mt-Metis	1.033	1.041	1.040	1.031	1.050	1.048



(a) 2x 8-core Xeon

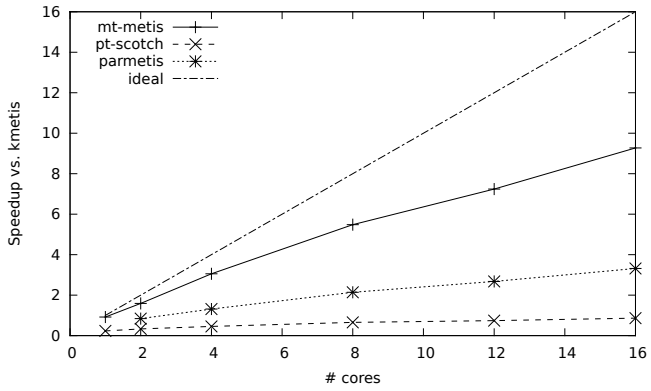


(b) 4x 8-core Opteron

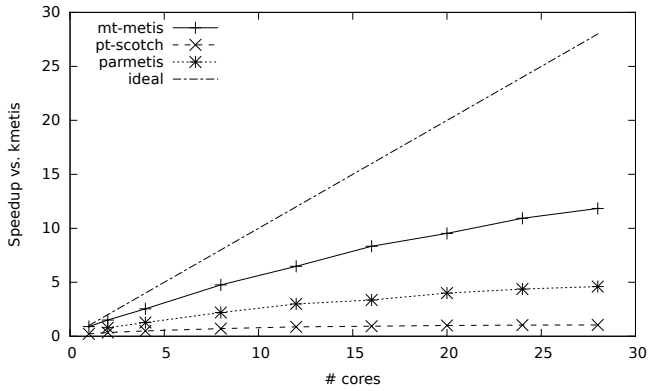


(c) 8x 4-core Opteron

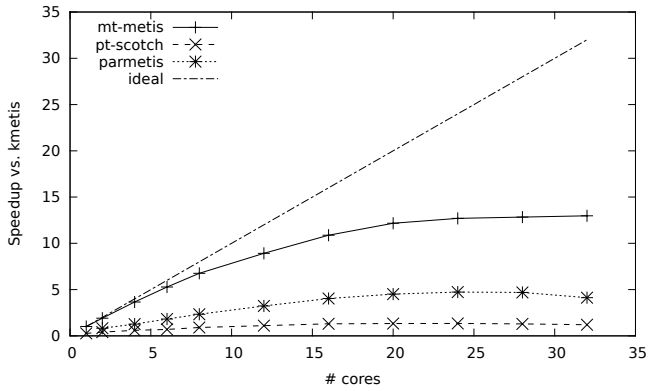
Figure 5.5: Mean speedup of the threading approaches on the three different systems.



(a) 2x 8-core Xeon



(b) 4x 8-core Opteron



(c) 8x 4-core Opteron

Figure 5.6: Mean speedups of the distributed memory partitioners and *mt-Metis* on the three different systems.

Chapter 6

Extensions to Shared Memory Multilevel Graph Partitioning

In this chapter, we present algorithmic improvements to the *mt-Metis* multithreaded graph partitioning framework and experimentally evaluate their effectiveness. We show that these modifications significantly improve performance of *mt-Metis* on modern architectures and graphs. Specifically, in this chapter we present the following:

- An efficient two-hop matching scheme which works well on graphs with highly skewed degree distributions without sacrificing performance or quality on graphs with more uniform degree distributions (2.0× geometric mean improvement for graphs with skewed degree distributions).
- Implementation level coarsening optimizations (1.6× geometric mean improvement for coarsening).
- An improved initial partitioning parallelization formulation (1.8× geometric mean improvement for initial partitioning).
- A method of performing parallel refinement that greatly reduces inter-core communication (2.5× geometric mean improvement for uncoarsening).

These improvements cumulatively result in speedups of $1.5 - 11.7\times$ and a geometric mean improvement of strong scaling by 82%, while preserving partition quality on 20 graphs from a variety of domains.

6.1 Algorithmic Improvements

Multilevel graph partitioning consists of three phases, each with its own set of operations. Improving the performance of a single operation or even phase is insufficient to make a meaningful impact on the runtime of graph partitioning. In this section we present algorithmic and implementation level improvements to all phases and operations of multilevel graph partitioning to significantly reduce the overall runtime.

6.1.1 Two-Hop Matching

Traditionally, vertices are aggregated together by finding maximal independent sets of edges to contract. This works well because it reduces the number of exposed edges on the graph (and subsequently exposed edge weight), and keeps the size of any coarse vertex from growing much faster than others. However, graphs with highly skewed degree distributions often contain only small maximal independent sets of edges. This causes the next coarser graph to be of similar size, and can cause many vertices to not grow in size at all between successive graphs.

To address this issue, we relax the constraint that two vertices being aggregated together must be connected via an edge. Instead, we allow two vertices to be aggregated together if they have a common neighbor. That is, if they are *two-hops* away on the graph. This has been investigated before in the context of finding vertex separators [147, 148] and graph clustering [139, 28].

To ensure we do not disrupt the quality achieved by traditional matching methods, we use two-hop matching as a secondary pass over the vertices after a maximal matching has been found. Our method assumes that each vertex in the graph has been visited, and that for each unmatched vertex, there exists no neighbor of that vertex for with which it is eligible to match. We group these unmatched vertices that are two-hops from each other into three classes: *leaves*, *twins*, and *relatives*.

Leaf vertices are of degree one, and if they share the same parent, it is desirable to aggregate them together. They are a subclass of twin vertices, but due to their prevalence in social networks and web graphs, using a special method to detect and match them is beneficial. Twin vertices are vertices which have identical neighbor lists. Relative vertices are vertices for which the shortest path between them is of length

two (they are two hops away), but do not have identical sets of neighbors. Relative vertices are the least desirable class to collapse, as doing so can hide good cuts in the coarser graphs. For these reasons we conditionally find and match each of these classes in the same order. If we have successfully matched over 75% of the vertices in the graph, we perform no two-hop matching. If after matching leaf vertices we still have not matched over 75%, we then perform twin matching. Finally, if this still does not yield a sufficiently large matching, relative vertices are then matched. Below we show that finding all three classes takes at most $O(n \log n)$, but is often linear in the number of unmatched vertices.

Finding Leaves

To find leaves to aggregate together in linear time, we iterate over our set L of unmatched vertices of degree one. For each vertex in L, we add its neighbor to the set of root vertices R. For each root vertex r, we keep track of all the unmatched leaf vertices $L_r \subset L$ we have processed that are incident to r (i.e., for each root vertex, L_r is the set of leaf vertices attached to it). Then for each root vertex r, we can match pairs of leaves in the list L_r, as we know they are two-hops from each other. As the sum the size of the set of unmatched leaf vertices L plus the size of the set of root vertices R cannot exceed the number of vertices in the graph, matching leaf vertices takes at most $O(n)$ time.

Finding Twins

Twin vertices are the most expensive vertices to aggregate together. To minimize this cost, we limit the maximum degree of vertices we consider for twins to 64 (though different values may be more desirable depending on graph characteristics and computational resources). We first sort all of our prospective twin vertices into buckets by degree. As we know this degree is of a bounded range (2 through 64), this sorting can be done in linear time via radix sort. We then sort each bucket using the vertices' neighborhoods as keys. As we have bounded the size degree of these vertices, we can compare two adjacency lists in linear time, giving us a $O(n \log n)$ complexity. During this sorting process we remove and match two vertices when their adjacency lists are equal. To further speedup this process, we first generate a hash of each vertex's adjacency list, and only perform the comparison based sort on vertices with equal hashes.

Finding Relatives

Finding pairs of relative vertices to match can be done using the same process as finding leaf vertices. However, because these vertices can be of degree larger than one the size of the set of the root vertices is no longer bounded by the number of candidates. Instead it is bounded by the number of edges incident to the candidates and by the total number of vertices in the graph. This makes the complexity of finding relative vertices of $O(n)$.

6.1.2 Coarsening Optimizations

During contraction we must translate adjacency lists from fine vertices to coarse vertices, and merge adjacency lists of vertices that have been aggregated together. From a matrix standpoint, this involves merging columns and rows of the adjacency matrix together. In our previous work [22], the approach here was to use a hash table to accumulate values for each coarse adjacency list. This ensured that when performing random accesses into the hash table, it resided in cache and reduced latency. For graphs with large maximum degree, a dense vector was used instead to avoid collisions in the hash table, but incurring the cost of latency associated with DRAM accesses.

For graphs with skewed vertex degree distributions, this is undesirable as the majority of the vertices have adjacency lists which can be merged in a hash table with few collisions. We can determine how many coarse vertices we will generate during aggregation. We then do a pass over the coarse vertices to be generated and calculate an upper bound on the degree of each coarse vertex (the sum of the degrees of fine vertices). We assign low degree vertices numbers increasing from zero, and high degree vertices numbers decreasing from the number of coarse vertices. This ensures during contraction we can use a hash table for the set of low degree vertices, and a dense vector for the set of high degree vertices where it is actually necessary.

During both aggregation and contraction, most of the memory accesses are through indirection arrays. In order to reduce the effects of latency, we use software prefetching. In aggregation, this consists of prefetching the locations of the match vector for neighbor vertices. During contraction, we prefetch the location of the coarse vertex mapping for the vertices in the adjacency lists.

6.1.3 Cache Oriented Initial Partitioning

The past approach for creating the initial partitioning relied on the fact that the coarsest graph was relatively small, and thus the amount of work required to create a partitioning was small. In this case, it is better to let several threads create initial partitionings via recursive bisection independently, avoiding synchronization overheads. However, this parallelism in the initial partitioning phase is limited to the number of partitionings to be created.

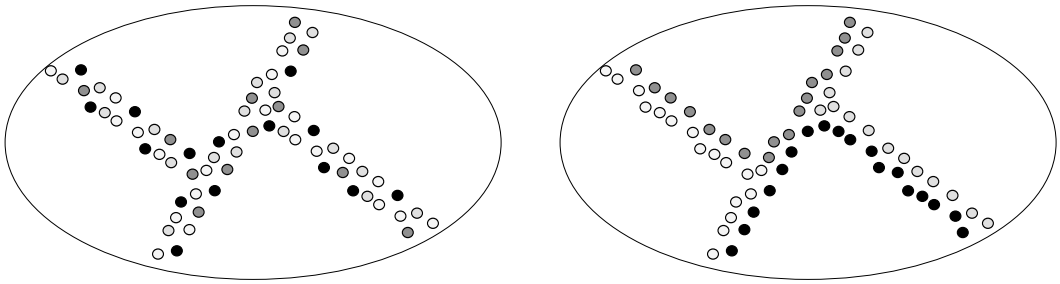
Our new method instead conditionally chooses to split the threads into independent groups to reduce inter-core communication. If the coarsest graph is large enough with respect to the number of threads, the threads will cooperatively work together to create the initial bisection. The threads will then split into two groups and recursively partition each half of the graph.

However, if the size of the coarsest graph is small enough with respect to the number of threads, the threads then break up into several groups, and each group independently generates a partitioning of the graph. We create our groups based on thread IDs which we bind to CPU cores, with the goal of creating groups that do not cross processor boundaries, thus making good use of shared caches and minimizing communication distance.

6.1.4 Boundary Migration

Vertices are statically assigned to threads during refinement for two reasons. First, multiple iterations of refinement are performed making it beneficial for threads to operate on the same set of boundary vertices in each iteration so as to promote data re-use. Second, the task size is exceedingly small, just a single vertex, and the overhead of task scheduling would dominate the runtime (using a larger task size would not guarantee that more than one vertex in a task would be on the boundary and require work).

In Chapter 5, threads performed refinement on the vertices they were assigned at graph generation (or input for the first level). This resulted in significant core-to-core communication. As seen in Figure 6.1(a), boundary vertices can be scattered among threads. Any time a vertex is moved, vertices owned by other threads must be updated. Handling these updates asynchronously means a lot of time is wasted processing small



(a) (b)

Figure 6.1: The different shades of vertices are assigned to different threads. The original assignment is shown in (a), where vertices in the boundary of the same partition may be assigned to many different threads. The migrated assignment is shown in (b), where boundary vertices of each partition have been assigned to a single thread.

messages from other threads during refinement and handling these updates in large batches or synchronously at the end of each iteration can result in extra work being performed in the form of suboptimal or discarded moves.

To address this issue, we introduce the notion of *boundary migration*. During the projection step of uncoarsening, we change the thread assignment of boundary vertices, so that rather than each thread owning vertices scattered throughout the boundary, each thread owns a relatively continuous chunk of boundary vertices as seen in Figure 6.1(b). We change the assignment of only boundary vertices so as to minimize the cost of this migration. Partitions are assigned to threads via hashing, and the boundary vertices are migrated to the threads to which their partitions were assigned.

To perform this migration, we create t buckets to place vertices in, where each bucket corresponds to the partitions assigned to each of the t threads. Each thread counts the number of boundary vertices that it owns at the start of projection destined for each bucket. A global prefixsum is computed such that each thread knows the starting index at which to insert its boundary vertices into the buckets. This is then followed by the threads copying their boundary vertices into the buckets. Once all threads have finished copying their boundary vertices into the buckets, each thread then retrieves the boundary vertices in the bucket corresponding to the partitions it was assigned.

Throughout all iterations of the current level of refinement a thread is responsible

for moving and updating the vertices which it was received in this process. When a vertex is pulled into the boundary, it is assigned to the thread that owns the partition in which it resides.

For the case where the number of threads is significantly less than the number of partitions, we assign multiple threads to a partition. We can then assign a vertex on the boundary in this partition to one of the partition's threads based on the opposing partition to which the vertex is most connected. When a vertex internal to a partition is pulled into the boundary (e.g., one of its neighbors was moved to another partition) it is assigned to one of the partition's threads via hashing. This hashing is done rather than assigning the vertex to the thread that pulled it into the boundary, as two or more threads may concurrently pull the same vertex into the boundary by moving its neighbors.

6.2 Experimental Methodology

The graphs used in the following experiments are a combination of scientific meshes, road networks, and non-linear programming matrices. Their details are listed in Chapter 3. The runtimes presented in the following sections are the mean of ten runs of the partitioners using different random seeds. We used an Intel Xeon E5-2699 v3 processor based system for the experiments. The system consists of two processors, each with 18-cores running at 2.3 GHz (a total of 36 cores) with 45 MB L3 cache and 64GB memory. The system is based on the Haswell microarchitecture and runs Redhat Linux (version 6.5). All our code is developed using C and is compiled using the GNU GCC version 4.8.3, using the O3 optimization flag. For comparison we used KaHIP version 0.71*c* from `http://algo2.iti.kit.edu/documents/kahip/index.html`, PT-Scotch version 6.0.4 from `http://gforge.inria.fr/projects/scotch/`, and ParMetis version 4.0.3 from `http://cs.umn.edu/~metis`.

6.3 Results

In this section we first evaluate our algorithmic improvements individually. We then evaluate the net effect of our algorithmic improvements. We will refer to *mt-Metis*

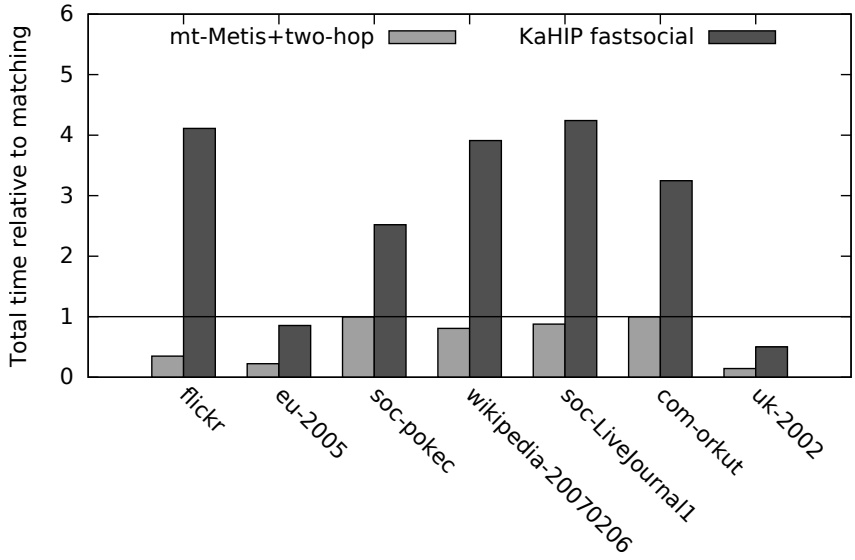


Figure 6.2: Two-hop matching and KaHIP's LP-based aggregation compared to *mt-Metis*, run serially and $k = 64$.

with these algorithmic improvements from Section 6.1 as *mt-Metis*-opt in the following experiments. Finally, we compare *mt-Metis*-opt to other parallel partitioners.

6.3.1 Coarsening

Aggregation

Figure 6.2 shows results of running *mt-Metis* serially with two-hop matching and KaHIP using the `fastsocial` configuration which uses size-constrained label propagation based aggregation. The runtimes are normalized to that of *mt-Metis* without two-hop matching running serially. As can be seen, allowing two-hop matching significantly reduces runtime, up to 7.0× for `uk-2002`, and a geometric mean for these seven graphs of 2.0×, as it allows the number of vertices in the graph to reduce by almost half at each level. This impacts not only the amount of work done in coarsening, but also the amount of work done in uncoarsening as well. The amount of time spent in initial partitioning is also reduced, as the size of the coarsest graph, G_s is smaller due to coarsening not exiting early.

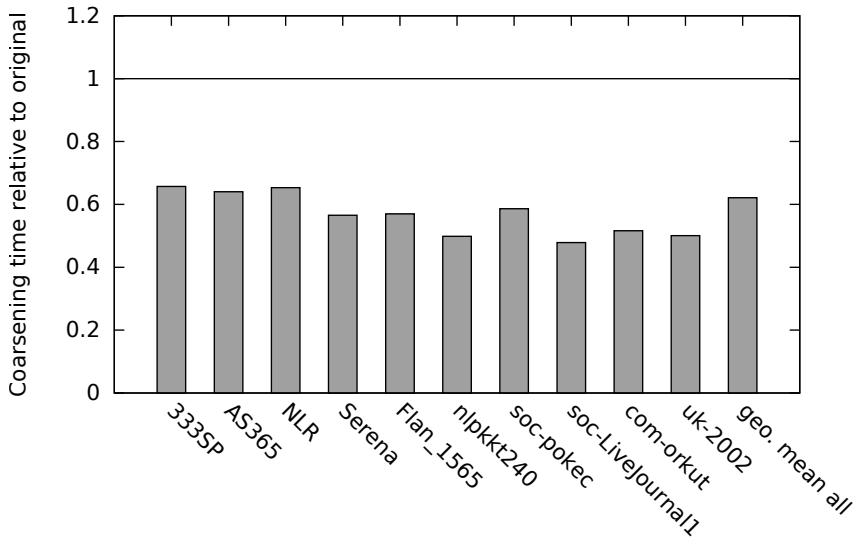


Figure 6.3: Coarsening runtime reduction due to optimizations (geometric mean for all 20 graphs), using 36 threads and $k = 64$.

The speedup from two-hop matching also brought with it an improvement in quality, decreasing the geometric mean of the number of cut edges by 3.2%. KaHIP's label propagation based aggregation allows it to detect a larger structures while coarsening, and does a better job leaving low-cut areas of the graph uncontracted. This resulted in a 14.4% lower geometric mean edgecut for the seven graphs than two-hop matching. This is largely a result of KaHIP finding good partitionings of `uk-2002` with half the edgecut of *mt-Metis*. The graph `uk-2002` has a strong cluster structure (few inter-cluster edges), and being able to detect those edges was crucial to finding small cuts on this graph. However, the amount of work associated with just a few iterations of label propagation far exceeds that of matching (and two-hop matching), causing KaHIP to have a geometric mean runtime 4.4× higher than *mt-Metis* with two-hop matching.

Implementation Level Optimizations

The results of our coarsening optimizations are shown in Figure 6.3, for the ten graphs on which they had the largest impact. The geometric mean for all 20 graphs is shown on the right. Our optimizations resulted in a geometric mean speedup of 1.6× for all

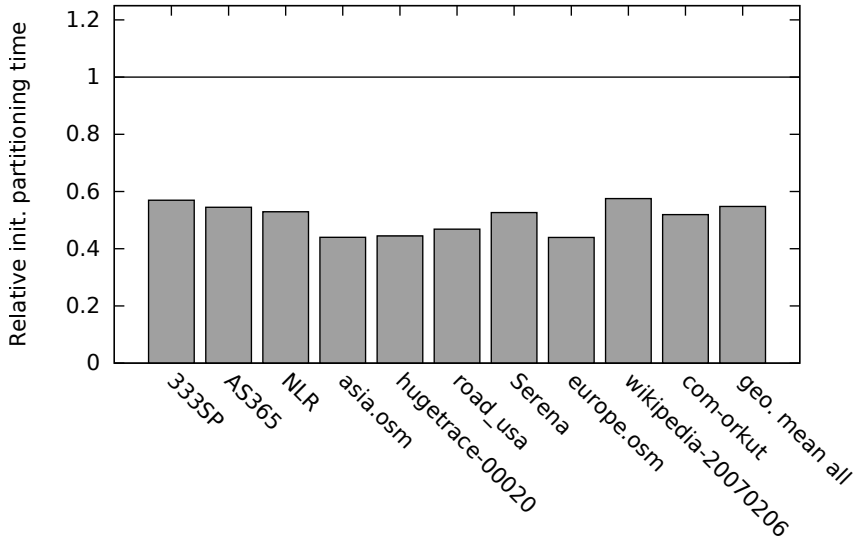


Figure 6.4: Cache oriented initial partitioning compared with independent initial partitioning (geometric mean for all 20 graphs), using 36 threads and $k = 64$.

20 graphs. Software prefetching resulted in large gains for the denser mesh-style graphs where we had a sufficient number of edges per vertex with which to look ahead. For the larger network style graphs, our two part contraction using both a hash table and a dense vector, played a large role in achieving near 2× speedups.

6.3.2 Initial Partitioning

The runtime of the new parallel formulation of initial partitioning for ten graphs is shown in Figure 6.4, and the geometric mean for all 20 graphs is shown on the right. The mean reduction in runtime was 45%, or a 1.8× speedup. While the semi-cooperative creation of initial partitionings means an increase in overhead, the increased parallelism more than made up for it. While the largest decrease in initial partitioning time was achieved on the sparse graphs as using all 36 threads did not result in cache conflicts, the largest impact on total running time was for the case where the coarsest graph was relatively dense. For example, for `com-orkut` which had a relatively dense coarsest graph, the total runtime decreased by 21%, and for `Flan_1565`, which had a relatively sparse coarsest graph, the total runtime decreased by less than 4%.

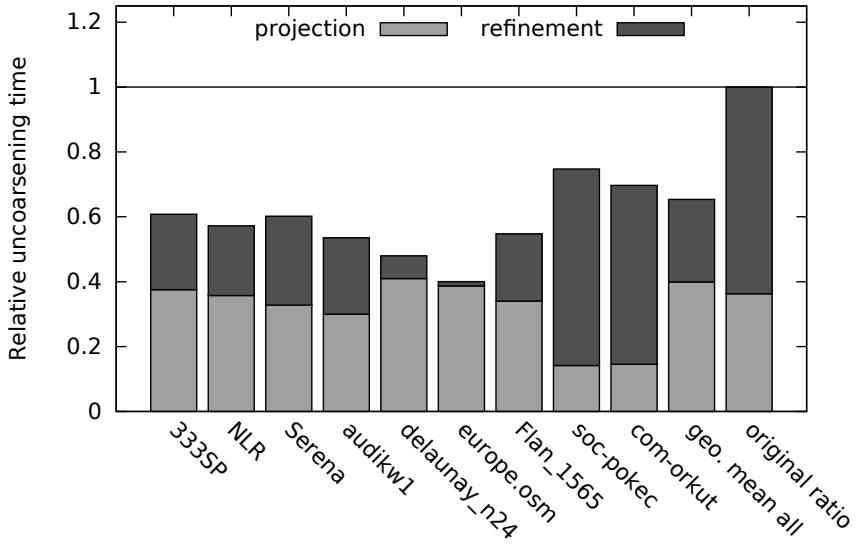


Figure 6.5: Uncoarsening runtime using boundary migration compared to static assignment (geometric mean is for all 20 graphs), using 36 threads and $k = 64$.

6.3.3 Uncoarsening

In Figure 6.5, we show the effects on runtime of migrating boundary vertices on ten of the graphs. The geometric mean for all 20 graphs is shown on the right. The time spent in refinement dramatically decreases when vertices are migrated, the geometric mean decreased by 2.5×. This is because the amount of updates that need to be communicated between threads dramatically decreases, and decisions regarding vertex movement are more likely to be made with up-to-date information and as a result moves are less likely to be undone in a later iteration.

Projection however, because it now includes the time it takes to migrate the boundary vertices, had its geometric mean runtime increase by 10%. This changed the percentage of time spent in projection from making up 36% of uncoarsening to 61%. The net effect of boundary migration reduced the geometric mean runtime of the uncoarsening phase when using 36 threads by 35%, and by up to 60% for the road network `europe.osm`. Because road networks tend to have very sparse cuts, the cost of communication between threads plays a significant role in the runtime of refinement where there is little useful work done. By migrating boundary vertices (of which there are very

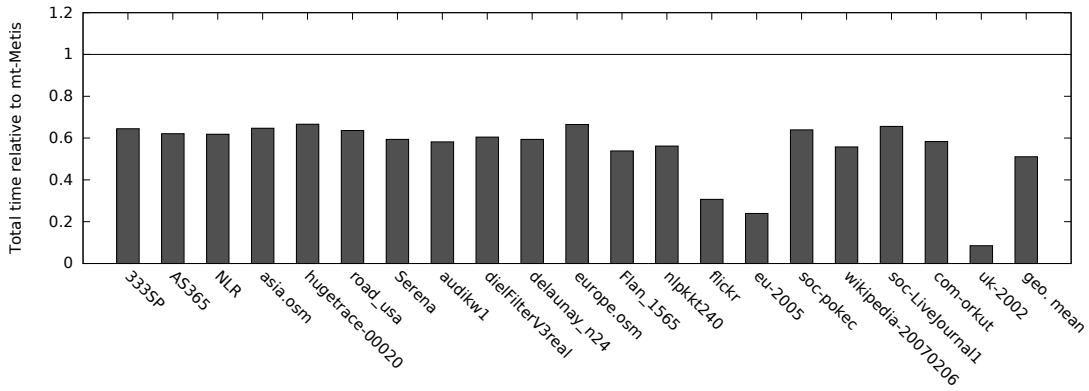


Figure 6.6: Comparison of runtime of *mt-Metis*-opt with *mt-Metis*, using 36 threads and $k = 64$.

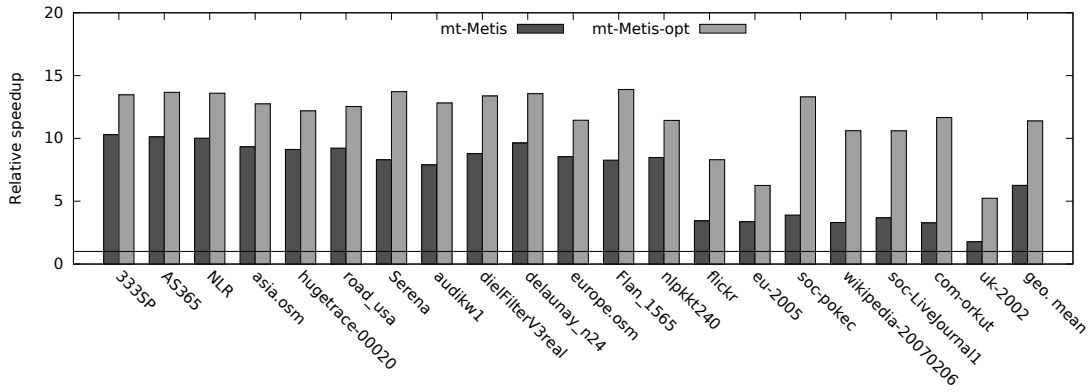


Figure 6.7: Comparison of strong scaling of *mt-Metis*-opt with *mt-Metis*, using 36 threads and $k = 64$.

few), we minimize this communication, which has a large impact on the runtime.

6.3.4 Overall Improvements

We present the net effects of our improvements in Figure 6.6, where we compare the runtime of our algorithmic improvements in *mt-Metis*-opt with *mt-Metis*. The geometric mean reduction in runtime was 49%, or a performance increase of 1.96×.

For the 20 graphs a range of performance improvement of $1.5 - 11.7\times$ was observed. The top of this range was achieved on `uk-2002`. This is largely due to the improved coarsening from two-hop matching, but was also influenced by large gains from our

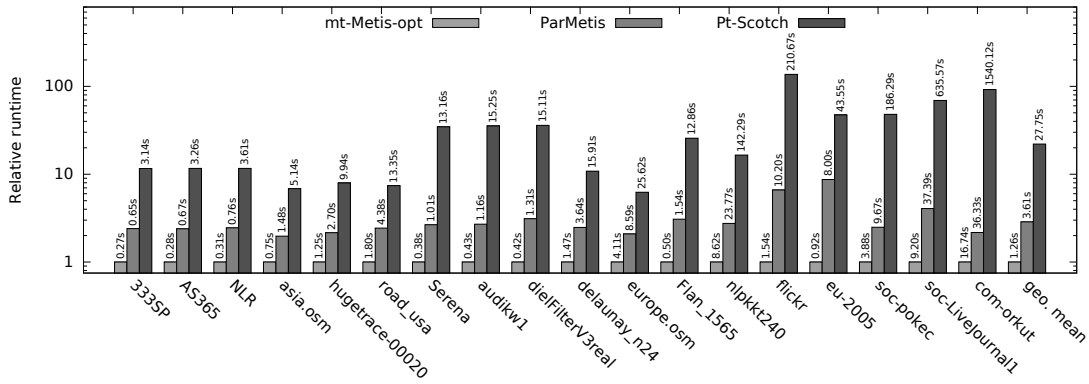


Figure 6.8: Comparison of modified *mt-Metis* with other partitioners, using 36 threads/processes and $k = 64$. Runtimes are relative with respect to the runtime of *mt-Metis*-opt. Absolute runtimes in seconds are shown above the corresponding bars.

coarsening optimizations and restructured initial partitioning. The geometric mean cut for the twenty graphs remained relatively unchanged with our algorithmic improvements (0.7% higher for *mt-Metis*-opt, largely due to higher edgecuts on road networks).

Our improvements not only made a significant difference in runtime, but also in terms of strong scaling, as shown in Figure 6.7. Where previously *mt-Metis* achieved a geometric mean speedup of 6.3× using 36 threads, held back in part by poor scaling on skewed degree distribution graphs, with our changes *mt-Metis*-opt scales to 11.4×. This is an improvement of 82%. For the skewed degree distribution graphs, two-hop matching shifts much of the runtime into the coarsening phase, which tends to scale the better than the other phases where there is a large amount of work per thread with little synchronization. Furthermore, our changes to initial partitioning and uncoarsening, increased the scalability of the remaining operations. This is evident when looking at the still substantial speedups for the graphs with non-skewed degree distributions.

In Figure 6.8, we compare the *mt-Metis*-opt with ParMetis and Pt-Scotch. The graphs `wikipedia-20070206` and `uk-2002` are not included in this experiment as ParMetis and Pt-Scotch ran out of memory while attempting to partition them. A parallel version of the KaHIP partitioner has not been released, and as such we do not compare against it here.

The geometric mean runtime of *mt-Metis*-opt was 2.9× lower than ParMetis for the

18 graphs. Pt-Scotch's geometric mean runtime was 7.6× higher than that of ParMetis, largely due to its use of recursive bisection, which requires roughly $\log k$ iterations through the multilevel paradigm (which is six for $k = 64$).

The largest difference of runtime between *mt-Metis*-opt and the distributed partitioners was on the graph `eu-2005`, where *mt-Metis*-opt was 8.7× faster than ParMetis. This large difference in runtime was due to ParMetis' inability to coarsen the graph. ParMetis was forced to stop coarsening at $344,515$ vertices, where as *mt-Metis*-opt coarsened `eu-2005` down to $6,779$ vertices before starting the recursive bisection in initial partitioning. The smallest difference was on the graph `asia.osm`, where *mt-Metis*-opt was 2.0× faster than ParMetis. This graph is very sparse with an average degree of slightly more than two, and has an extremely small boundary on 64-way partitions (only 0.01% of the vertices were on the boundary). As a result of these properties, 90% of the time was spent in coarsening and the projection step of uncoarsening, and our coarsening optimizations were targeted at graphs where the work associated with the edges was much greater than that of the work associated with the vertices. For the denser mesh-style graphs, `dielFilterV3real` and `Flan_1565`, *mt-Metis*-opt was 3.1× faster than ParMetis, largely due to our coarsening optimizations and the much smaller refinement time resulting from boundary migration.

Chapter 7

High Quality Shared Memory Refinement

In this chapter we present a new shared memory parallel method for directly refining a k-way partitioning that incorporates hill-climbing. Our new method, Hill-Scanning, produces solutions of equal quality to Pairwise FM [112] and Multi-Try FM [114]. We show that our method runs in $O(kn/p + (m/p)\log n)$ time, where k is the number of partitions, n is the number of vertices, m is the number of edges and p is the number of threads. We present strong scaling results with up to 24 threads and show that it achieves speedups of $5.7 - 16.7\times$, while exhibiting only 0.52% increase in edgecuts.

7.1 Hill-Scanning Refinement

In this section, we present our Hill-Scanning refinement algorithm. We first describe a simplified version of this algorithm in Section 7.1.1, for refining 2-way partitions serially. We then show how to extend this algorithm to the k-way setting in Section 7.1.2. We then present the full version of our algorithm, for refining k-way partitionings on shared memory parallel architectures in Section 7.1.3.

Algorithm 1 Hill-Scanning Refinement

```
1:  function HillScan(G,P,φ)
2:      q ← priority queue
3:      repeat
4:          insert boundary vertices into q
5:          while |q| > 0 do
6:              v ← pop(q)
7:              if positive gain for v then
8:                  move v
9:              else
10:                 h ←BuildHill(v,G,P,φ)
11:                 if h ≠ ∅ then
12:                     move h
13:                 end if
14:             end if
15:         end while
16:     until no vertices are moved
17:     return P
18: end function
```

7.1.1 Two-Way Hill-Scanning

The Hill-Scanning (HS) algorithm for use in refining a 2-way partitioning is outlined in Algorithm 1. It takes three input arguments, the graph G, the current partitioning P, and the maximum hill size ϕ.

Each iteration works as follows. First, all of the boundary vertices, those with edges connecting them to the opposing partition, are inserted into a priority queue. The gain associated with moving a vertex is used as the priority. This gain for the vertex v is the sum of the weight of the edges connecting it to the opposing partition minus the sum of the weight of the edges connecting it to the partition in which it resides:

$$gain = d_{ext}(v) - d_{int}(v).$$

Vertices are extracted from this queue and are considered for moving. If the gain

Algorithm 2 Hill Building

```
1:  function BuildHill(v,G,P,φ)
2:      h ← ∅
3:      q ← priority queue
4:      insert v into q
5:      while |q| > 0 and |h| < φ do
6:          u ← pop(q)
7:          h ← h ∪ {u}
8:          if moving h is beneficial then
9:              break
10:         end if
11:         Add all u ∈ Γ(u), ∈ A to q
12:     end while
13:     if moving h is beneficial then
14:         return h
15:     else
16:         return ∅
17:     end if
18: end function
```

associated with moving a vertex v is positive, and moving v would not violate the balance constraint, v is moved to the opposing partition, and its neighbors are updated in the priority queue. Vertices with zero gain will still be moved if it improves the balance of the partitioning.

If the gain associated with moving a vertex v is not positive, we attempt to build a hill rooted at v. If we identify a hill rooted v with a positive gain, we move the hill to the opposing partition. If a vertex is moved by itself or as part of a hill, it is locked in place and prevented from moving for the rest of the iteration.

The intuition behind this algorithm is that we want to avoid the *move-and-revert* process used in KL/FM like algorithms, and instead improve the partitioning at each successive state. By attempting to move each vertex individually before searching for a hill, we are able to overlap fine-grain and coarse-grain partitioning improvements.

The hill building function used by Algorithm 1 is shown in Algorithm 2. This function is what separates the hill-scanning algorithm from the Greedy algorithm. How far we explore looking for a hill, the maximum hill size ϕ, determines the trade-off between runtime and quality. Because our algorithm is for use in the multilevel setting, we can use a relatively small value for ϕ, based on the intuition that very large hills will likely have been moved during a coarser round of refinement.

The function `BuildHill` starts by initializing an empty hill, and inserting the root vertex v into the hill priority queue. The hill is then grown by extracting vertices from the priority queue. When a vertex u is extracted from the top of the priority queue, it is added to the hill. If the gain associated with moving the entire hill is positive, the loop exits and the hill is returned. Otherwise, the neighbors of u are added to the priority queue. If the hill reaches the maximum allowable size and would not result in a positive gain if moved, it is discarded and an empty set is returned.

To keep the runtime down, each time an edge is traversed when building a hill, it is marked as *traveled* for that direction. During each iteration, an will be traversed at most once in each direction. This prevents vertices from being repeatedly inserted into the priority queue as hills are discarded. Furthermore, we observed that hills built earlier in a refinement pass were far more likely to be moved than those later in the pass. As such, add an early exit when $\sqrt{b(V)}$ hills have been built and discarded, where $b(V)$ is the number of vertices on the boundary (i.e., vertices with edges connecting them to vertices in the opposing partition).

In the 2-way setting, our hill-scanning algorithm is functionally similar to CLIP/CDIP [113] searching for hills in a localized area at a time. However, because it identifies hills before moving them, we can extend it to the k-way and parallel settings.

7.1.2 k-Way Hill-Scanning

As with the two-way version of the algorithm, in the k-way Hill-Scanning algorithm we insert all boundary vertices into a priority queue. To accurately order vertices in the priority queue based on their gain, we would need to track:

$$gain = \max_{P_i \in P'} d_{P_i}(v) - d_{int}(v),$$

where P' is the set of partitions for which moving v to would not violate the balance constraint. This however would require updating vertex priorities frequently as partition weights and vertex connectivities change.

Instead we use the approximate gain associated with moving the vertex v out of its partition as the priority. We model this as the external edge weight divided by the square root of the number of external partitions minus the internal edge weight:

$$priority = \frac{d_{ext}(v)}{\sqrt{\pi(v)}} - d_{int}(v).$$

For vertices that are only connected to a single external partition, this accurately models their priority. For vertices connected to more than one external partition, this favors vertices connected to fewer partitions while not over penalizing vertices connected to too many partitions.

Building the hill in the k-way setting requires several modifications from the 2-way setting. We can no longer model the gain associated with a hill as the sum of the external edge weights minus the sum of the internal edge weights, as the external edge weights can be split among multiple partitions.

To accurately model the gain associated with a hill h in partition A as we build it, we keep a vector W of length k, which stores the connectivity of the hill all partitions. Each time we add a vertex v to the hill, we scan its adjacency list, adding the weight of the edges to the corresponding entries in W. For edges connecting v to the hill, instead of adding the entry for A in W, we subtract the weight from it. Thus, after adding each vertex the gain associated with moving the hill to partition B is $W_B - W_A$.

While Multi-Try FM [114] is also works on k-way partitions, once it moves its seed vertex for a trial, it moves all subsequent vertices in that trial to the same partition. This can lead to hills being moved to a partition of lesser gain, or not moved at all. This can cause Multi-Try FM to require more iterations to migrate the hill to its most desirable partition. Because HS identifies a hill before it decides where to move it, it ensures the hill will be moved to the partition of maximum gain.

7.1.3 Parallel k-Way Hill-Scanning

The move-and-revert strategy of KL/FM like algorithms is difficult to parallelize due to the need of a serialized order of moves. While methods have been proposed for running

FM on independent subgraphs [25], these require some degree of pre-partitioning.

Because Hill-Scanning does not use a move-and-revert strategy, we can parallelize at a coarse level. The movement of vertices in hill-scanning is the same as in Greedy refinement, so we model the parallelization of hill-scanning after the method [22] for parallelizing Greedy refinement on shared memory architectures.

In parallel Hill-Scanning, each refinement iteration is split into two phases: upstream and downstream. At the start of each iteration, we assign each partition a random unique integer label. These labels are then used to induce an acyclic flow on the partition-graph (a graph in which each partition is represented by a single vertex). During the upstream phase, vertices are only considered for moving to partitions with higher labels than the label of the partition in which they currently reside. In the downstream phase, vertices are only considered for moving to partitions with lower labels. Updated information regarding the state of the partition and vertex locations are communicated asynchronously between threads via message queues.

Because we do not use any thread synchronization primitives when building hills it is possible for vertices to be simultaneously added to the hills of different threads that overlap. If the overlapping hills move to different partitions, and may cause a hill segment to move to a partition of lower gain. However, this race condition occurs rarely, and when it does, the misplaced hill segment will be moved to its correct location in the next iteration as it will be identified as a small hill.

7.1.4 Complexity

In this section, we analyze the complexity of Hill-Scanning refinement. We start by establishing the complexity of 2-way Hill-Scanning in Section 7.1.4, and work our way up to showing that the full parallel k-way version of our algorithm runs in $O(kn/p + (m/p)\log n)$ time.

Complexity of 2-Way Hill-Scanning

In two-way hill-scanning for a graph with n vertices and m edges, we will insert and extract up to $O(n)$ vertices in the priority queue. Because we lock each vertex after moving it, we move at most $O(n)$ vertices and perform at most $O(m)$ neighbor updates

for these moves. We can then say the cost of moving vertices and selecting single vertices to move is $O(m \log n)$.

When we build a hill, we mark each edge as traveled, for each direction. This means that each edge may be traversed at most twice during an iteration, and thus the number of insertions and updates to the priority queue is bounded by the number of edges. As the priority queue can have at most all of the vertices in the graph in it, the cost of building hills is $O(m \log n)$. Combining this with the cost of selecting and moving vertices, we see that the total complexity of HS is $O(m \log n)$ per iteration.

Complexity of k-Way Hill-Scanning

In terms of complexity, k-way Hill-Scanning differs from two-way hill-scanning in two places: determining the best partition which to move a vertex to, and building hills. When determining which partition to move a vertex (or a hill) to, we can consider at most k partitions. This add an additional $O(kn)$ term to the complexity.

When building hills in k-way Hill-Scanning, can at most perform an operation on the vector W per edge, adding an addition $O(m)$ term to the complexity. This is hidden by the larger terms we previously derived for operations on the priority queues. Thus, combining the cost of considering to which partition a vertex/hill is to be moved and the cost of operations on the priority queue, the cost of k-way Hill-Scanning becomes $O(kn + m \log n)$.

Complexity of Parallel k-Way Hill-Scanning

We assign n/p vertices and their incident edges to each thread, and for the rest of this analysis we assume the assigned incident edges per thread is m/p. Thus the maximum number of vertices inserted into a given thread's priority queue is n/p, making the cost of performing an update $(\log(n/p))$. As each thread has m/p edges associated with its n/p vertices, each thread will need to update its vertices at most m/p times. This gives a complexity of $O(kn/p + (m/p)\log(n/p))$ for selecting and moving vertices.

We know the total cost of building hills is bounded by $O(m \log n)$, due to the marking of edges as traveled. While threads can traverse edges and visit vertices other than their own while building hills, two threads cannot traverse the same edge in the same direction. The means we have at most m edge traversals split among p threads. This gives

hill building a complexity of $O((m/p)\log n)$, which is larger than the $O((m/p)\log(n/p))$ term for selecting and moving vertices. Finally, parallel k-way Hill-Scanning has a complexity of $O(kn/p + (m/p)\log n)$.

7.2 Experimental Setup

7.2.1 Data

For these experiments we used 30 graphs from those listed in Chapter 3. The first set of graphs (`wing` through `auto`) are all graphs with greater than 100 thousand edges from the Graph Partitioning Archive [34]. The second set of graphs are the non-zero patterns of some of the largest matrices from the University of Florida Sparse Matrix Collection [1].

7.2.2 System Configuration

These experiments were run on a machine with 2× 12 core Xeon E5-2680v3 @ 2.5GHz processors and 64GB of memory. The operating system was CentOS 6.6, running the Linux kernel version 2.6.32. The code was compiled using GCC 4.9.2.

7.2.3 Implementation

We implemented HS, and the other refinement algorithms detailed below in the *mt-Metis* multithreaded graph partitioning framework. The version used for these experiments is *mt-Metis* 4.4. The matching scheme used is *Heavy Edge Matching* [92]. Each refinement scheme terminates when an iteration completes without any moves, or a maximum of 8 iterations have been performed.

Greedy

We used the implementation of parallel Greedy refinement in *mt-Metis* for these experiments, described in Chapter 5.

FM

We implemented a boundary-only variant of the Fiduccia-Mattheyses [97] (FM) algorithm for use in recursive bisection. Due to FM's serial nature, only a single thread is used for refining each bisection. After each bisection, the threads split into two groups to perform the remaining bisections on each half of the graph. We used a limit of 100 vertices being moved without gain before FM reverts back to the minimum bisection.

KPM

For our implementation of k-way pairwise FM (KPM), we split the 8 iterations into two local iterations and four global iterations. Parallelism is expressed by refining pairs of partitions in parallel in each global iteration, which has a maximum concurrency of $k/2$. We used a limit of 32 vertices being moved without gain before FM reverts back to the minimum bisection between a pair of partitions.

MTFM

For our serial experiments, we implemented Multi-Try FM [114] (MTFM). To keep the runtime down, we used a limit of 16 vertices being moved without gain before the seeded FM reverts back to the best state, and the next seed vertex is selected.

HS

We implemented the parallel k-way version of the Hill-Scanning (HS) algorithm we detailed Section 7.1.3. For the k-way and parallel experiments, we used a maximum hill size ϕ of 16, and a maximum number of discarded hills of 64.

7.3 Results

First, we examine the effects of varying the maximum hill size ϕ in the Hill-Scanning algorithm on runtime and quality in Section 7.3.1. This is followed by a comparison of HS with Greedy, FM, and KPM refinement schemes in Section 7.3.2. Finally, in Section 7.3.3, we perform strong scaling experiments to examine the effects of parallelization on runtime and quality of HS.

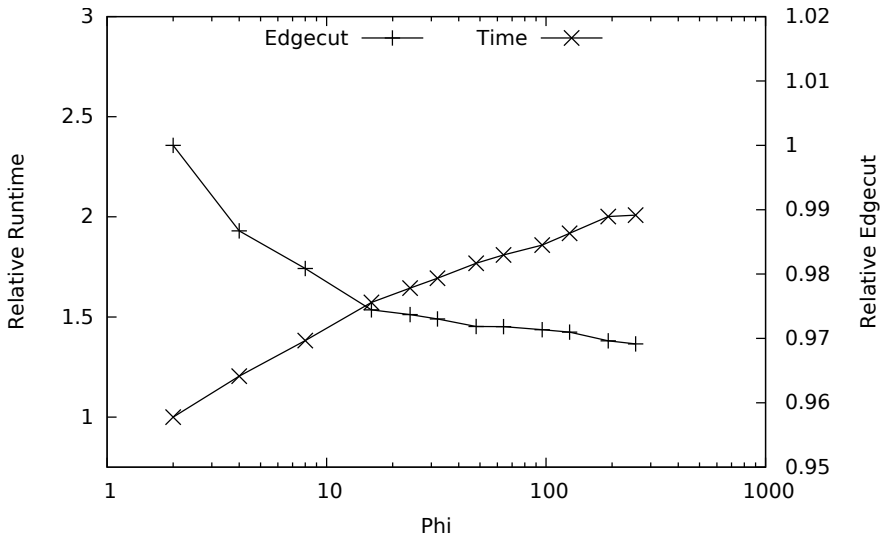


Figure 7.1: Effects of varying ϕ on `Flan_1565`.

7.3.1 Maximum Hill Size

In Figure 7.1, we present the effects of varying ϕ, the maximum size of a hill, on the quality and runtime of refinement for the graph `Flan_1565`. As the hill size increases, the further inside of a partition is explored, and thus the runtime increases relatively steadily. However, at a hill size of 16, the increase in quality slows as the larger hills are less likely to be moved compared to smaller ones. Due to this, we use a value of 16 for ϕ for the remainder of the experiments presented here as it gives a good balance of both quality and speed.

7.3.2 k-way Refinement

The performance of the different refinement schemes, run serially, is compared in Table 7.1. The results presented are the geometric mean of 25 runs. Runtime includes the entire multilevel process: coarsening, initial partitioning, and uncoarsening. As expected, Greedy refinement is the fastest method, but also results in the worst quality (highest edgecuts). It is faster than the other methods not only because it does fewer calculations per vertex moved, but also because it tends to moves fewer vertices. The Hill-Scanning algorithm was the second fastest method, and the multilevel process using HS took only

27.1% longer than using Greedy. However, HS resulted in the lowest geometric mean edgecut of the refinement schemes, and had the smallest geometric mean edgecut on 14 of the 30 graphs. Multi-Try FM had a geometric mean edgecut for the 30 graphs 1% higher than HS, and had the smallest mean edgecut on three of the graphs. Both HS and MTFM focus on making localized k-way moves, which is why their behavior when run serially is similar.

RB-FM did well on the smaller graphs, averaging the lowest edgecut on two of the graphs. The fact that optimal bisections applied recursively do not correspond to an optimal k-way cut played less of a role on these smaller graphs. KPM found solutions of similar quality of HS and MTFM, and had the lowest mean edgecut for eleven of the 30 graphs, most of which were the larger graphs. This is because on the larger graphs, more vertices could be moved between a pair of partitions without violating the balance constraint. KPM however, was also the slowest method, especially on the larger graphs. This high runtime is the result of running FM on each connected pair of partitions, which for partitionings with relatively dense partition connectivity can be exceedingly expensive.

7.3.3 Parallel k-way Refinement

Figure 7.2 shows the strong scaling of the HS algorithm on a 24 core machine. HS achieves speedups between 5.7× and 16.7×, with a geometric mean of 9.3× using 24 threads. This compares to mean speedup for the Greedy algorithm of only 2.7×. Because HS performs more work per iteration (more vertices visited and more vertices moved), the overheads associated with parallelizing refinement are a smaller fraction of the runtime.

Figure 7.3 shows the relative edgecut as the number of threads is increased. The resulting edgecut changed slightly for most of the graphs as the degree of parallelism was increased. It increased the most for `ldoor`, going up by 2.7%, and decreased the most for `Flan_1565`, decreasing 1.7%. However, after eight threads, these changes largely plateau as we increase the number of threads to 24. The geometric mean increase across all ten graphs was only 0.52%, demonstrating the stability of parallel HS.

We compare the total runtime of the multilevel process using the four parallel refinement schemes in Figure 7.4. The geometric mean runtimes for RB-FM, KPM, and

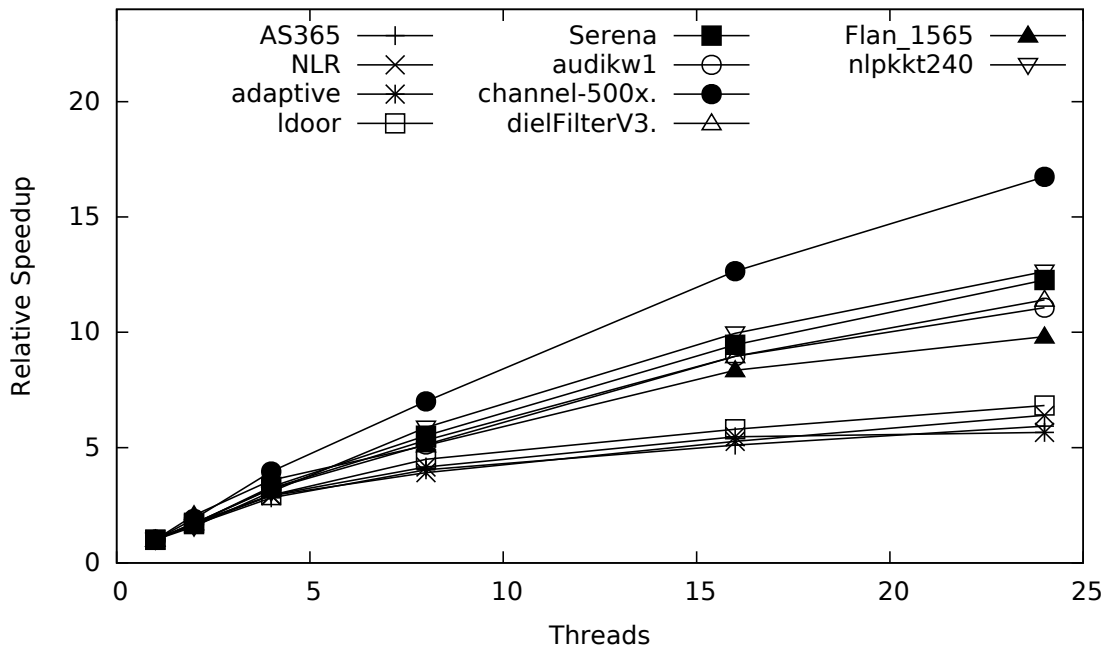


Figure 7.2: Strong scaling of Hill-Scanning with respect to speedup.

HS using 24 threads are plotted relative to the runtime of Greedy using 24 threads to create 64-way partitions. HS closes the gap with Greedy refinement with this degree of parallelism, averaging only 17% longer total partitioning time. Not only are RB-FM and KPM slower when run serially, they both have limited parallelism. RB-FM must operate serially when making the first bisection, and does not fully express p parallelism until after the first $\log p$ bisections. While KPM can express up to $k/2$ way concurrency via edge coloring the partition-graph G_P, many of the resulting colors will have less than $k/2$ edges when G_P is not a complete graph, further limiting the degree of parallelism.

Figure 7.5 shows the geometric mean edgecut of RB-FM, KPM, and HS relative to Greedy using 24 threads. HS had a geometric mean edgecut 6.3% lower than Greedy. This is 3.4% and 1.9% lower than RB-FM and KPM respectively. This shows that not only is HS extremely fast and able to scale well, but it also produces the best quality among parallel refinement schemes.

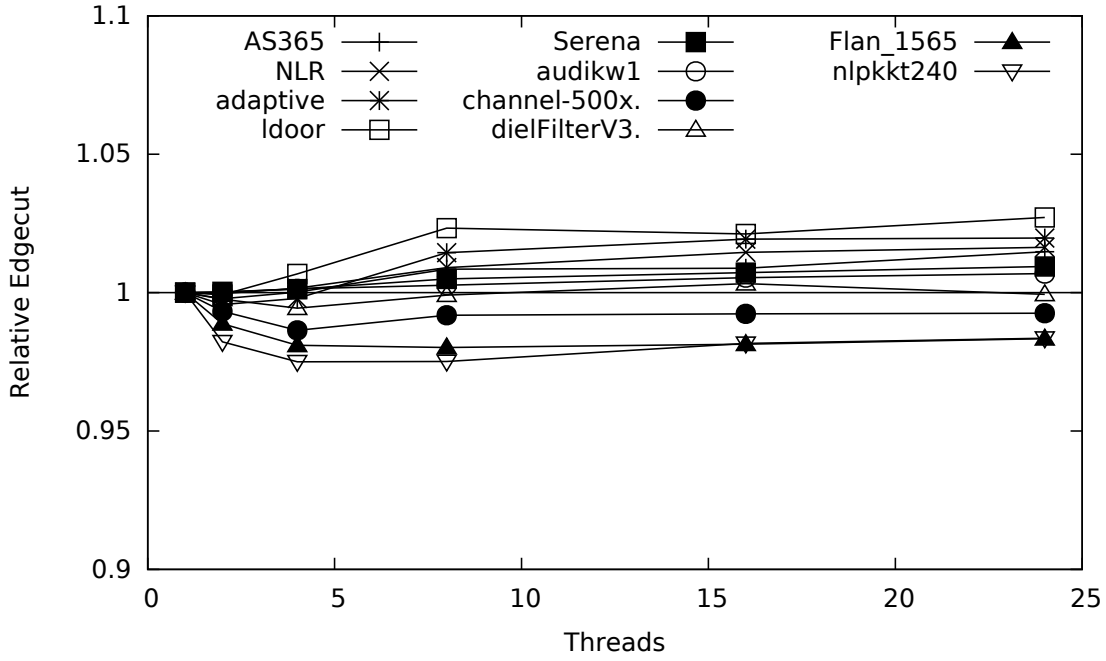


Figure 7.3: Strong scaling of Hill-Scanning with respect to relative edgecut.

Table 7.1: Edgecut and serial runtimes for 64-way partitions with a 0.03 balance constraint.

	Greedy		RB-FM		KPM		MTFM		HS	
Graph	Edgecut	Time (s)	Edgecut	Time (s)	Edgecut	Time (s)	Edgecut	Time (s)	Edgecut	Time (s)
t60k	2,565	0.085	2,433	0.202	**2,378**	0.481	2,445	0.108	2,401	0.109
wing	9,727	0.146	9,074	0.309	8,783	1.351	8,772	0.248	**8,592**	0.220
fe_pwt	9,451	0.091	9,124	0.204	8,776	0.485	8,929	0.132	**8,775**	0.126
fe_body	5,710	0.079	**5,236**	0.221	5,289	0.429	5,458	0.097	5,352	0.096
vibrobox	54,046	0.205	54,799	0.293	53,405	0.994	53,028	0.232	**52,835**	0.247
finan512	11,500	0.148	**10,710**	0.395	11,388	0.803	11,632	0.297	11,350	0.183
bcsstk33	116,821	0.237	117,623	0.280	114,427	0.725	**114,168**	0.257	114,322	0.273
bcsstk29	63,929	0.127	62,348	0.266	**61,149**	0.485	63,432	0.138	62,413	0.148
brack2	29,805	0.150	28,721	0.408	**28,104**	1.113	28,555	0.219	28,414	0.217
fe_ocean	27,312	0.198	23,011	0.586	**22,826**	2.032	23,385	0.364	22,896	0.349
fe_tooth	39,987	0.169	39,009	0.484	38,219	1.365	38,261	0.273	**38,032**	0.265
bcsstk31	67,391	0.168	65,103	0.386	**63,852**	0.953	65,470	0.215	64,568	0.225
fe_rotor	52,616	0.199	51,553	0.667	50,279	1.953	50,815	0.336	**50,251**	0.326
598a	63,413	0.225	63,346	0.745	**60,299**	2.303	60,756	0.401	60,440	0.370
bcsstk32	107,911	0.170	102,943	0.535	**102,382**	1.021	104,614	0.215	103,550	0.220
bcsstk30	190,499	0.193	187,906	0.546	**183,650**	0.968	186,719	0.237	185,576	0.257
wave	95,460	0.274	95,142	0.952	91,778	2.859	90,803	0.522	**90,515**	0.485
144	88,039	0.264	87,836	0.964	84,254	2.827	84,245	0.490	**83,643**	0.455
m14b	109,570	0.335	108,677	1.411	103,996	3.484	104,563	0.625	**103,878**	0.576
auto	191,400	0.681	191,933	2.841	183,624	6.652	181,215	1.355	**180,367**	1.203
AS365	54,767	3.154	52,993	15.519	51,943	15.984	50,740	3.822	**50,356**	3.669
NLR	60,287	3.538	58,430	17.249	57,437	17.553	55,775	4.319	**55,378**	4.134
adaptive	48,634	4.452	44,364	20.789	44,149	24.593	42,651	5.911	**42,344**	5.330
ldoor	439,153	1.624	420,011	9.834	**414,884**	5.846	421,377	1.771	415,463	1.816
Serena	1,852,915	3.140	1,869,282	15.720	1,813,903	31.495	1,762,200	5.079	**1,760,964**	4.828
audikw1	2,945,167	3.269	2,976,115	17.174	2,838,997	23.877	**2,830,537**	4.941	2,835,015	4.951
channel-500x.	1,356,670	6.633	1,274,048	31.187	1,305,570	64.801	1,274,548	12.781	**1,260,250**	11.258
dielFilterV3.	2,442,877	3.652	2,432,023	20.173	**2,321,037**	26.516	2,335,146	5.272	2,321,886	5.054
Flan_1565	2,463,890	4.376	2,392,417	25.932	**2,317,798**	19.220	2,344,077	5.412	2,335,178	5.846
nlpkkt240	10,303,386	65.096	10,326,787	297.996	10,171,102	156.130	**9,751,898**	96.392	9,763,379	108.195
Geo. Mean	105760.6	0.540	102440.1	1.795	100294.6	3.457	100542.3	0.791	**99634.8**	0.764

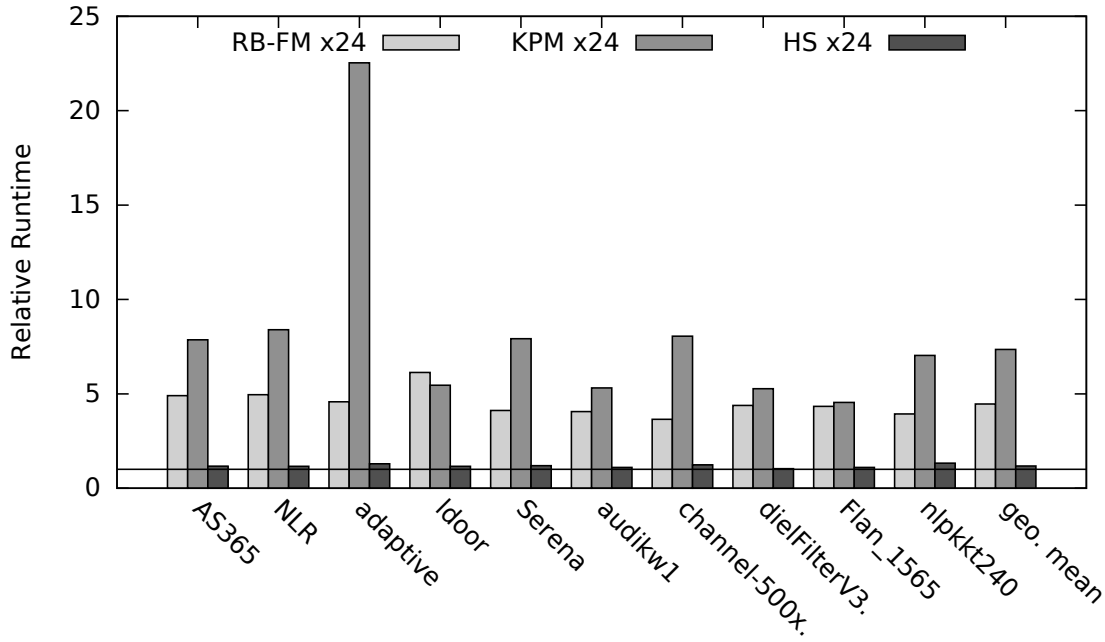


Figure 7.4: Relative partitioning time of refinement schemes compared to Greedy refinement using 24 threads.

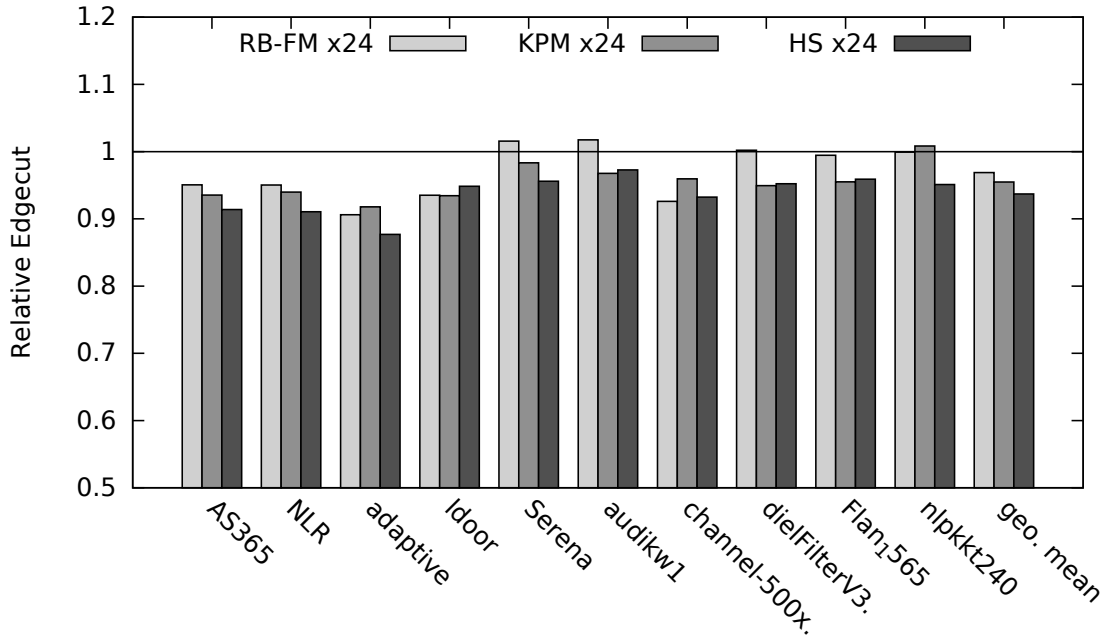


Figure 7.5: Relative edgecut of refinement schemes compared to Greedy refinement using 24 threads.

Chapter 8

Sparse Matrix Ordering

In this chapter we present shared memory parallel algorithms for generating vertex separators and using those vertex separators to generate a fill reducing ordering via nested dissection in parallel. We build on the algorithms in Chapter 5 creating edge separators using the multilevel paradigm on shared memory architectures. We adapt these algorithms for vertex separators and introduce a new method for refining a vertex separator in parallel while making minimal sacrifices in terms of separator size. We introduce specialized task scheduling to maximize cache efficiency for the nested dissection problem. We achieve up to 10× speedup on 16 cores, while producing orderings with only 1.0% more fill-in and requiring only 0.7% more operations than the serial *ND-Metis* [20]. This is 1.5× faster, 3.7% less fill-in, and 14.0% fewer operations than *ParMetis* [20].

8.1 Methods

Our methods for performing nested dissection build upon our work on multithreaded multilevel graph partitioning discussed in Chapter 5. We use the same parallelization and coarsening strategies. Each thread is assigned a set of vertices and their associated edges, and is responsible for the computations on them.

8.1.1 Vertex Separators

The generation of vertex separators differs from edge separators in the initial partitioning and uncoarsening phases.

Initial Separator Selection

A widely used method of generating a vertex separator from an edge separator is to find a vertex cover of the set of cut edges [83]. Because we apply refinement to the separator, we instead take all boundary vertices as the initial separator of the coarsest graph G_s, and let refinement thin the separator and possibly move it away from the boundary set of vertices. We repeat this process several times and select the minimum balanced separator. As these separators are generated and refined independently, the process is inherently parallel. As the input graph is the same across the generation of different separators, waiting until G_s is sufficiently small so as to fit into shared cache is desirable.

Separator Refinement

After the current separator is projected from G_i to G_{i-1}, it is refined. Refinement of a vertex separator consists of moving vertices from the separator S into either partition A or partition B. If a vertex being moved is connected to vertices on the opposite side of the separator, those vertices are then pulled into the separator. The reduction in separator size from moving vertex $v \in S$ to A is

$$gain = \eta(v) - \sum_{u \in \Gamma(v) \cap B} \eta(u). \tag{8.1}$$

FM Refinement: The Fiduccia-Mattheyses refinement (FM) algorithm [97], as applied to the vertex separator problem [123], works as follows. First, priority queues for moving vertices out of the separator to either partition are initialized and filled with vertices in S. The priority of vertices in these queues is determined by equation (8.1). Vertices are selected from either priority queue in order of gain, except when one partition is overweight, in which case the vertex at the top of the priority queue for the lower weight partition is selected. Once a vertex is selected, it is moved out of the separator, and its neighbors in the opposite partition are pulled into the separator. If the neighbors being pulled into the separator have not been moved yet in this refinement pass, they are added to the priority queue. Once both priority queues are emptied, the best observed state is restored. To reduce runtime, this process is terminated early if a

certain number of moves past the best state have been made. Keeping track of the best state and reverting to it, makes the FM algorithm inherently serial.

Greedy Refinement: The greedy algorithm moves vertices through the separator to one side at a time. This is done so that at any given moment, the current state of the separator is valid. First, the lowest weight side of the separator is selected as the side to which all moves will be made in the first pass. Then, each thread adds the vertices it owns that are part of the separator to its own priority queue, using equation (8.1) for the priority. Each thread makes a local copy the current partition weights which it uses to keep track of moves and enforce the balance constraint. These weights are periodically synchronized with the global weights as moves are made. While this makes it possible for refinement to violate the balance constraint if enough vertices are moved before partition weights are synchronized, it is unlikely as it is desirable for the balance constraint on vertex separators in nested dissection to be large [148]. In practice we have not observed Greedy refinement to cause imbalance.

Each thread then extracts vertices from its priority queue. If the vertex can be moved out of the separator without violating the balance constraint, and has a positive gain associated with it, it is moved. The neighboring vertices that the thread owns have their connectivity information updated and are added to the separator as applicable. Messages are sent to the threads owning the remote vertices to notify them of the move.

Once the queue is empty, or the gain associated with moving the top vertex is negative, the thread waits for the other threads to finish. The thread then reads its messages, and updates its vertices accordingly. Finally, the threads synchronize once more, and the process repeats with the other side selected. While efficient, this method often results in lower quality than the serial FM algorithm as it cannot break out of local minima.

Segmented FM Refinement: Because we want the improved quality that results from breaking out of local minima, one possible solution is to have threads perform FM on internal vertices (vertices which are not connected to vertices owned by another thread). We will refer to this approach as Segmented FM (SFM), which for these internal vertices works the same as the serial FM algorithm and allows us to break out of local minima in parallel. External vertices, those that have neighbors belonging to other threads, are prevented from moving out of the separator. This ensures that as

long as each thread maintains a valid separator for its vertices, the global separator will also be valid. Each thread saves its best locally observed state, and independently reverts back to it at the end of each pass.

For this method to be effective, each thread must have a large number of internal vertices and few external vertices. We accomplish this by creating a k-way edge separator of the graph using the techniques described in Chapter 5 as a pre-processing step. While this increases the runtime, it is a parallel step and scales well. Furthermore, this pre-partitioning improves data locality, which is particularly beneficial for nested dissection where we can use a single pre-partitioning for the entire process. We select a value for k that is several times larger than the number of threads, and assign partitions to threads via hashing so that each thread owns vertices in several locations of the graph. This is done so that many of the threads will own vertices that will be part of the separator, and the work during refinement will be distributed across multiple threads. We found using a value of k that is five times the number of threads to be effective.

While this method allows us to find high quality local separators, the inability to move external vertices prevents the separator from moving significantly. For more than a few threads, this can have a significant impact on separator size as is shown in Section 8.3.

Greedy with Segmented FM Refinement: Both Greedy refinement and SFM refinement have their advantages and disadvantages. Greedy refinement's ability to move both internal and external vertices allows it to move the separator freely, but it cannot break out of local minima. SFM refinement can break out of local minima for a thread's internal vertices, however external vertices anchor the separator in place, limiting the improvement. As quality is one of our primary concerns, these disadvantages make both Greedy and SFM refinement unattractive options on their own.

For this reason, we investigated a hybrid refinement strategy by overlapping Greedy and SFM refinement passes. The first greedy pass thins the separator and moves it to a local minima. Next, the SFM pass moves the sections of the separator on internal vertices out of the local minima. The next Greedy pass then allows the external vertices to catch up with the moved internal ones. This is shown in Figure 8.1. This process repeats until neither the Greedy pass nor the SFM pass move any vertices. This provides an effective refinement scheme that can break out of local minima and move external

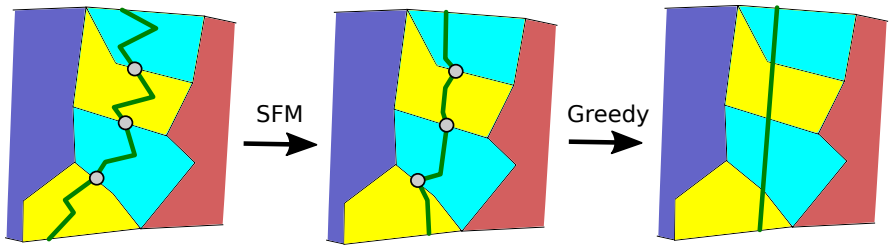


Figure 8.1: First, the projected separator is refined with SFM improving the separator for partition interiors, then Greedy refinement is applied, improving the portion of the separator crossing partition boundaries.

Algorithm 3 Parallel Nested Dissection

```
1:  function ND(G)
2:      if Number of threads is greater than 1 then
3:          A, B, S ← vertex separator of G, in parallel
4:          P_A ← half the threads call ND(A)
5:          P_B ← half the threads call ND(B)
6:      else
7:          A, B, S ← vertex separator of G, serial
8:          Add ND(A) to work pool
9:          Add ND(B) to work pool
10:         Wait for ND(A) and ND(B) to finish
11:     end if
12:     return {P_A, P_B, S}
13: end function
```

vertices in parallel, without leading to an invalid separator.

8.1.2 Nested Dissection

Our parallel nested dissection algorithm is described in Algorithm 3. At the first level, all threads call the function `ND`. The threads then induce a vertex separator cooperatively, and use this to split the graph into parts A and B. The threads then split into two groups, with one group recursing on A and the other recursing on B, generating the

orderings P_A and P_B respectively. This repeats until each thread group contains only a single thread. Each thread then spawns tasks for processing A and B, and adds them to the work pool. Once both A and B have been ordered, the ordering of G is computed by placing A first, B second, and S last. When $|A|$ is small enough, it is ordered via the Multiple Minimum Degree algorithm [127] discussed in Chapter 3. This is omitted from Algorithm 3 for simplicity.

Task Scheduling

Splitting the recursive calls on the graph parts A and B into parallel tasks allows us to dynamically balance the computational load. However, we need to effectively utilize the cache to overcome memory bandwidth and latency bottlenecks. The task tree of nested dissection has several properties that we want to keep in mind when scheduling the tasks. 1) The lower a task is on the tree (the earlier it is generated), the larger the graph that is associated with it. 2) The graph associated with a given task is a subgraph of the graph associated with its parent's task, thus the best cache use is achieved by having a task processed immediately after its parent.

To maximize our cache use, we propose a task scheduling scheme specifically for the nested dissection problem, that takes advantages of these properties. Our scheduling scheme operates on two levels. Each thread maintains a local list of tasks that it generates. It processes the tasks in its list in Last-In First-Out order to ensure that whatever subgraph is currently cached is used by the next scheduled task as often as possible. When a thread runs out of tasks in its own list, it steals tasks from neighboring threads in First-in First-out order (the largest tasks). This not only ensures stolen tasks have enough work associated with them to achieve cache re-use, also ensures that the stolen tasks are the ones least likely to have their associated graph resident in another thread's cache. In Section 8.3.3 we compare this scheduling scheme against the generic scheme implemented in the OpenMP runtime.

8.2 Experimental Methodology

The experiments in this chapter were run on a HP ProLiant BL280c G6 with 2x 8-core Xeon E5-2670 @ 2.6 GHz system with 64GB of memory. We used Intel C Compiler,

Table 8.1: Size of Vertex Separators

	auto	NLR	med_fe	delaunay_n24	large_fe	nlpkkt240
FM (serial)	**2,133**	**1,811**	2,166	3,507	6,421	156,564
Greedy	2,277	1,918	2,281	4,167	6,717	148,665
SFM	2,985	2,264	5,882	4,302	12,430	262,243
Greedy+SFM	2,205	1,821	**2,071**	**3,492**	**6,024**	**146,523**

version 13.1, and the GNU GCC compiler 4.9.2. The algorithms evaluated here are implemented in *mt-Metis* 0.4.0, which is available from `http://cs.umn.edu/~lasalle/mtmetis`. We will refer to the new vertex separator and nested dissection functionality as *mt-ND-Metis* in the following experiments. For comparison, we also used *Metis* [20] version 5.1.0 (referred to in the experiments as *ND-Metis*) from `http://cs.umn.edu/~metis`, *ParMetis* [122] version 4.0.3 from `http://cs.umn.edu/~metis`, and *Scotch* [21] version 6.0.3 from `http://www.labri.fr/perso/pelegrin/scotch`.

The results presented for vertex separators are the geometric means from 25 runs using different random seeds. The results presented for nested dissection are the geometric means from 10 runs using different random seeds.

8.3 Results

8.3.1 Vertex Separators

Table 8.1 shows the effect on separator size of the different refinement schemes. We compare the three parallel methods run with 16 threads to that of serial FM. SFM refinement resulted in large separators compared to that of serial FM, due to its inability to move external vertices. Greedy refinement did much better, finding separators only 6.1% larger than serial FM. The refinement scheme combining both Greedy and SFM refinement passes, produced separators of comparable size to FM, and for several graphs found slightly smaller separators on average. The number of external vertices that are prevented from being moved when trying to break out of a local minima in this scheme is quite small due to our pre-partitioning.

Table 8.2 shows the effect on runtime of the different refinement schemes. The

Table 8.2: Refinement Time in Seconds

	auto	NLR	med_fe	delaunay_n24	large_fe	nlpkkt240
FM (serial)	0.044	0.178	0.104	0.898	0.336	3.183
Greedy	0.048	0.091	0.071	0.130	0.181	1.251
SFM	**0.030**	**0.069**	0.068	**0.115**	0.185	**1.153**
Greedy+SFM	0.050	0.101	**0.062**	0.147	**0.134**	2.678

runtime of serial FM is included for comparison against the other three refinement schemes run with 16 threads. None of the parallel refinement schemes exhibit significant speedup over FM consistently. There are two reasons for this. First, refinement operates on a small portion of the graph, and requires frequent synchronization. Second, the parallel refinement schemes make more passes before they settle on a separator. This also explains why the Greedy+SFM scheme is sometimes faster than the SFM and Greedy schemes. It performs more work per pass than either Greedy or SFM, but settles on a separator in fewer passes.

Figure 8.2(a) shows the strong scaling of *mt-ND-Metis* generating vertex separators using up to 16 cores. The time shown includes the cost of pre-partitioning the graph, which is why there is a slowdown observed between one and two threads. The speedup achieved is largely dependent upon the size of the graph, and how effectively the amount of work between synchronization points can hide the parallel overhead. Looking beyond two threads, the larger graphs achieve speedups nearing 6× overall. Discounting the pre-partitioning time, the largest and third largest graphs exhibit super linear scaling with speedups over 17×. This is due to improved locality that comes from the pre-partitioning, and the extra cache available on the second processor. This shows the importance of having a well distributed graph, even on shared memory architectures.

8.3.2 Task Scheduling

Table 8.3 shows the percent improvement of our nested dissection task scheduling scheme, over that of the implementation schemes provided by ICC [145] and GCC [149]. Our scheme was on average 41.1% faster than the ICC scheduler and 40.6% faster than the GCC scheduler. This is because these schedulers are designed to handle tasks with

Table 8.3: Improvement over OpenMP Task Scheduling

	auto	NLR	med_fe	delaunay_n24	large_fe	nlpkkt240
ICC OMP	68.9%	38.3%	48.7%	30.4%	39.9%	25.9%
GCC OMP	62.2%	39.0%	60.2%	25.6%	40.0%	23.0%

varying properties, whereas our specialized scheduler takes advantage of the nature of the nested dissection task tree.

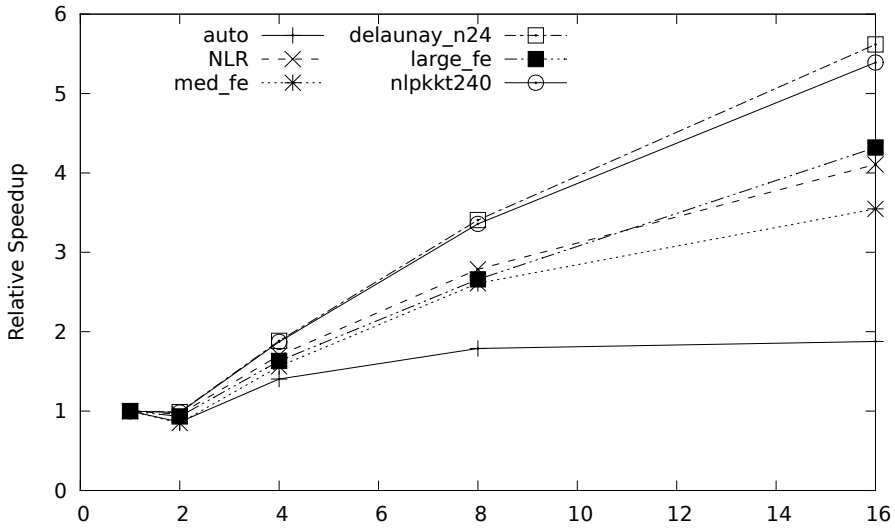
8.3.3 Nested Dissection

Figure 8.2(b) shows the strong scaling of *mt-ND-Metis* performing nested dissection. For the smallest graph, `auto`, the achieved speedup is limited to 3.3×, as the parallel overhead plays a significant role in the runtime. For the larger graphs, the different graph operations performed dominate the runtime and hide the parallel overhead. As a result, speedup of 6–10× is achieved on the other five graphs. We see a greater speedup here than on just vertex separators as the cost of performing nested dissection is significantly greater than that of creating a k-way edge separator, and better hide its added cost.

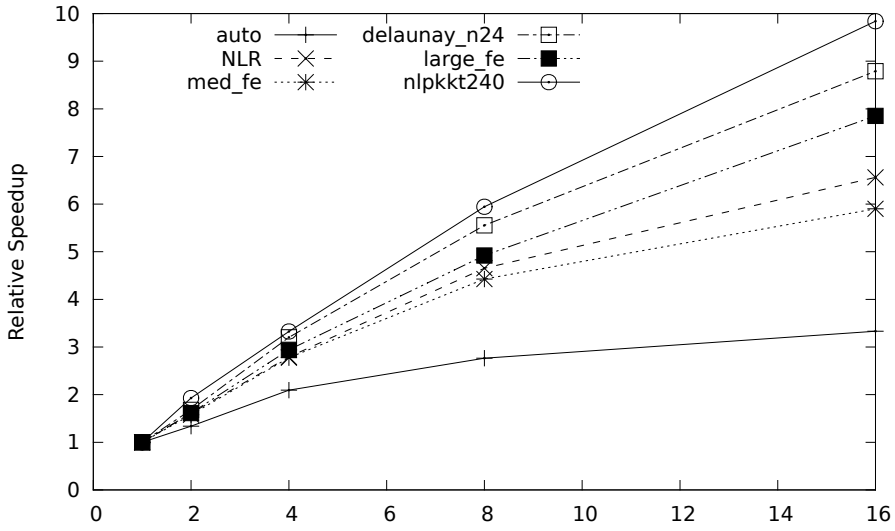
Table 8.4 compares the orderings of *mt-ND-Metis* with that of *ND-Metis*, *ParMetis*, and *Scotch*, in terms of number of non-zeros in the Cholesky factor and the operations required to compute it. The runtimes to generate these orderings are also included (excluding I/O, but including preprocessing). Making efficient use of the multicore system, *mt-ND-Metis* was on average 1.5× faster than the other two parallel methods, and 10.1× faster than the serial *ND-Metis*. The number of operations required by orderings produced by *mt-ND-Metis* were only 0.7% higher than those required by *mt-ND-Metis*, and 14.0% lower than those required by *ParMetis* or *Scotch*. The hybrid refinement of *mt-ND-Metis* enables these high quality results, close to that of *ND-Metis*. The high-speed parallel vertex separator generation during the low levels of the nested dissection tree coupled with the specialized task scheduling in the higher levels enables *mt-ND-Metis* to produce orderings the fastest for all datasets except the smallest.

Table 8.4: Comparison of Nested Dissection

	auto	NLR	med_fe	delaunay_n24	large_fe	nlpkkt240
ND-Metis						
Fill-in	**2.22e+08**	**2.05e+08**	2.88e+08	**7.24e+08**	1.61e+09	**1.98e+11**
Operations	**4.53e+11**	**1.25e+11**	3.83e+11	**7.39e+11**	4.57e+12	**1.93e+16**
Time (s)	7.94	51.82	39.26	248.83	184.58	1148.52
mt-ND-Metis 16 Threads						
Fill-in	2.31e+08	2.06e+08	**2.87e+08**	7.30e+08	**1.55e+09**	2.07e+11
Operations	5.06e+11	1.28e+11	**3.71e+11**	7.46e+11	**3.94e+12**	2.04e+16
Time (s)	1.44	**4.67**	**4.44**	**17.85**	**16.34**	**93.80**
ParMetis 16 Processes						
Fill-in	2.29e+08	2.13e+08	3.10e+08	7.58e+08	1.60e+09	2.17e+11
Operations	4.94e+11	1.52e+11	4.98e+11	9.40e+11	4.51e+12	2.30e+16
Time (s)	1.60	6.21	6.43	29.52	31.17	169.84
PT-Scotch 16 Processes						
Fill-in	2.52e+08	2.73e+08	3.84e+08	9.72e+08	1.93e+09	2.62e+11
Operations	5.89e+11	3.39e+11	8.70e+11	2.00e+12	8.57e+12	2.79e+16
Time (s)	**1.12**	5.83	7.46	26.82	39.33	678.65



(a) Vertex Separator Generation



(b) Nested Dissection

Figure 8.2: Strong Scaling of *mt-ND-Metis* on 16 Cores

Chapter 9

Shared Memory Multilevel Graph Clustering

In this chapter we present multilevel algorithms for effectively and efficiently generating a graph clustering which maximizes the modularity objective. The contributions of our work are:

- A method for efficiently contracting a graph for the modularity objective.
- An robust method for generating clusterings of a contracted graph.
- A modified version of boundary refinement for the modularity objective.
- Shared-memory parallel formulations of these algorithms.

We show that for a graph with n vertices and m edges, these algorithms have $O(m + n)$ time and $O(m + n)$ space complexities. We show that the shared memory parallel versions of these algorithms have a parallel time complexity of $O(m/p + n/p + k)$ where p is the number of threads and k is the number of clusters. To validate our contributions, we compare our implementation of these algorithms, *Nerstrand*, against the serial clustering tool *Louvain* [136] and the parallel clustering tools *community-el* [138] and *NetworKit* [140], and show that *Nerstrand* produces clusterings of equal or greater modularity and is 4.5–27.2 times faster than the methods that generate clusterings with competitive modularity. We also compare the quality of clusterings generated by the

serial version of *Nerstrand* against the results of the 10th DIMACS Implementation Challenge [48] on graph partitioning and graph clustering. The modularity of clusterings produced by *Nerstrand* are equal to or within only a few percentage points of the best clusterings reported in the competition while requiring several orders of magnitude less time. The parallel version of *Nerstrand* is scalable and extremely fast, clustering a graph with over 105 million vertices and 3.3 billion edges in 90 seconds using 16 cores.

9.1 Serial Clustering Methods

We investigate aggregation schemes to address the issue of coarsening graphs with power-law degree distributions in Section 9.1.1. We introduce a method for effectively generating initial clusterings of a coarsened graph in Section 9.1.2. We present a formulation of refinement for maximizing modularity in Section 9.1.3. We give a complexity analysis of these algorithms in Section 9.1.4, showing that they run in $O(m + n)$ time and $O(m + n)$ space.

9.1.1 Coarsening

We explored three different aggregation schemes: *matching* (MAT), *matching with secondary two-hop matching* (M2M), and *first choice grouping* (FCG). For all three aggregation schemes, we attempt to merge all vertices, which helps to prevent the skewed cluster sizes present in greedy agglomerative methods. These three schemes choose vertices to aggregate together by selecting the vertex u to aggregate v with that maximizes the function $Q_{merge}(v, u)$. This is the change in modularity that would result if v and u were clusters and were merged to form a single cluster. The change in modularity by merging v and u is

$$Q_{merge}(v, u) = Q_{\{v,u\}} - (Q_{\{v\}} + Q_{\{u\}}). \tag{9.1}$$

In the special case where $v = u$, $Q_{merge}(v, u) = 0$ (i.e., there is no change in modularity if a vertex is merged with itself).

Matching

Our standard matching algorithm (MAT) works similar to the Heavy Edge Matching discussed in Chapter 3. It visits each vertex v in random order and matches v with the unmatched neighbor $u \in \Gamma(v)$ for which equation (9.1) is maximized. If v has no unmatched neighbors, or equation (9.1) is below zero, v is matched with itself (aggregated by itself). Standard matching works well on graphs with near uniform degree distribution as matchings tend to be very large. However, as discussed in Chapter 3, for graphs with skewed degree distributions, which is often the case for graph clustering problems, standard matching is ineffective.

Two-Hop Matching

To address these matchings of small size in MAT, we applied the two-hop matching (M2M) technique discussed in Chapter 6 to the modularity maximization problem. This aggregation scheme works the same as MAT, except after attempting to match all vertices, unmatched vertices are revisited and possibly matched with other vertices two hops away (the shortest path between them is of length two). This works the same as MAT, There is no prioritization in finding vertices in two hop matching, and instead the first unmatched vertex is selected.

First Choice Grouping

The third option we explored for aggregating power-law graphs was to allow more than two vertices to be aggregated together at a time. The first choice grouping (FCG) scheme is based on the FirstChoice aggregation scheme originally used for contracting hypergraphs [93] and later applied to contracting simple graphs for the graph partitioning problem [107]. Our formulation differs from these earlier methods in that we not only consider the weight of the edge, but the current state of vertex groupings and the associated modularity gain.

An outline of this scheme is given in Algorithm 4. When searching for a vertex or vertices to aggregate the vertex v with, all of the neighbors $u \in \Gamma(v)$ are considered regardless of whether they have been matched/grouped already. If u is ungrouped, then its priority for grouping is determined using equation (9.1). If u belongs to the group

Algorithm 4 First Choice Grouping (FCG)

```
1: function FirstChoiceGrouping(G(V, E))
2:     Mark all v ∈ V as unmatched
3:     for all v ∈ V do
4:         if v is unmatched then
5:             c ← an empty group
6:             for all u ∈ Γ(v) do
7:                 if u is unmatched then
8:                     if Q_merge(v, u) > Q_merge(v, c) then
9:                         c ← u
10:                    end if
11:                else
12:                    g ← group of u
13:                    if Q_merge(v, g) > Q_merge(v, c) then
14:                        c ← g
15:                    end if
16:                end if
17:            end for
18:            Add v to the group c
19:        end if
20:    end for
21: end function
```

g, then the priority for adding v to that group is determined similarly, except the edges from g to v need to be summed, and $d(g)$ needs to be tracked.

9.1.2 Initial Clustering

Once coarsening is finished, we are left with the coarsest graph G_s, and need to determine the number of clusters k and create the clustering $C = \{C_1, C_2, \ldots, C_k\}$. We call this process *initial clustering*. Initial clustering is done with a direct clustering scheme, that is, a non-multilevel scheme that operates directly on G_s.

In G_s, each vertex is the result of collapsing together clusters of fine vertices during

coarsening. For this reason, we can use a relatively simple initial clustering scheme. Our initial clustering scheme works by setting each vertex to be a singleton cluster as in agglomerative clustering and then applying refinement as described in Section 9.1.3. This is similar to a single level of the Louvain method [136].

9.1.3 Uncoarsening

In the uncoarsening phase, we take the clustering of the coarsest graph, G_s, and use it as an estimate for a good clustering of the finer G_{s-1}. We then improve it for G_{s-1} finding a local maxima of modularity. This is repeated until the clustering is applied to, and improved for G_0. The process of applying the clustering of G_i to G_{i-1} is referred to as *projection*. The process of improving the clustering for G_{i-1} is referred to as *refinement*.

Projection

Projection in *Nerstrand* is done by propagating cluster information from the coarse vertices in G_{i+1} to the fine vertices in G_i. By keeping track of what fine vertices compose a coarse vertex, we can project a clustering of G_{i+1} to G_i, by assigning each fine vertex in G_i to the same cluster that its coarse vertex is assigned. Since we keep track of collapsed edge weight for each coarse vertex, and use them in computing cluster degrees, the modularity of the clustering does not change in projection.

Refinement

We developed two modularity based refinement methods: *Random Boundary Refinement*, and *Greedy Boundary Refinement*. These two methods differ only in the order in which they consider vertices for moving. Both methods visit only vertices that are connected via an edge to one or more vertices which reside in different clusters. These vertices are referred to as *boundary* vertices. Similarly, when considering moving a vertex, we only evaluate the gain associated with moving it to a cluster to which it is connected.

It is possible that moving a vertex to a cluster to which is not connected or moving a vertex that is not a boundary vertex could result in a positive gain in modularity. For this to occur, when moving the vertex $v \in C_i$ to the cluster C_j to which it has no

connection, the difference in the degree of C_i and the degree C_j must make up a larger fraction of the total edge weight in the graph than the fraction of v's edge weight that connects it to C_i:

$$\frac{d(C_i - \{v\}) + d(C_j)}{d(V)} > \frac{d_{C_i}(v)}{d(v)}.$$

We observed that when considering all vertices for movement to all clusters resulted in only a 0.06% gain in modularity, while taking over 16 times as long. Furthermore, Brandes et al. [150] showed that a clustering with maximum modularity does not include non-contiguous clusters.

The gain by moving a vertex from cluster C_i to cluster C_j is given by the combined change in the cluster modularities:

$$\Delta Q(v, C_j) = (Q_{C_i - \{v\}} + Q_{C_j + \{v\}}) - (Q_{C_i} + Q_{C_j}).$$

Note that if it leads to a positive gain in modularity, clusters can be completely emptied and removed during refinement.

If at least one vertex was moved while visiting all of the boundary vertices, another pass is performed. Refinement stops when no vertices are moved in a pass, or when a maximum number of passes has been made.

Random Boundary Refinement (RBR) visits the boundary vertices in random order. This has two advantages. The first is that we can visit all of the boundary vertices in linear time. The second is that it is stochastic, and we can perform it multiple times using the same input clustering with different random seeds to explore the solution space.

Greedy Boundary Refinement (GBR) first inserts the boundary vertices into a priority queue. Each vertex is then extracted from this priority queue and considered for moving to a different cluster. As the state of the clustering changes, the priority of the vertices remaining in the priority queue is updated. This ensures that we continually make the best available move for the current clustering state.

To accurately prioritize vertices for movement between clusters based on modularity gain, we would need to use:

$$\Delta Q = (Q_{C_i - \{v\}} - Q_{C_i}) + \arg\max_j (Q_{C_j + \{v\}} - Q_{C_j}). \tag{9.2}$$

This however, is a computationally expensive priority to maintain as the arg max part of the equation will change each time a vertex is moved to or from one of the clusters to which v is connected.

We decided instead to use a heuristic for the priority. This heuristic uses the modularity gain associated with removing the vertex from its current cluster only (the left side of equation (9.2)). Using this priority, boundary vertices are inserted into a priority queue. Vertices are then extracted from the priority queue and the modularity gains associated with moving the front vertex are evaluated fully.

When a vertex v is moved from C_i to C_j, we only update the priority of the vertices connected to it, even though all the priority of all vertices in C_i and C_j have changed. In our experiments we did not observe an increase in the modularity of clusterings if we kept the priority of all vertices up to date.

9.1.4 Complexity Analysis

The overall complexity for the serial algorithms in *Nerstrand* is the sum of its three phases:

1. Coarsening, $O(m+n)$, in Section 9.1.4.
2. Initial Clustering, $O(m+n)$, in Section 9.1.4.
3. Uncoarsening, $O(m+n)$ for RBR and $O(m \log n)$ for GBR, in Section 9.1.4.

Adding these we get an overall computational complexity of $O(m+n)$ (and $O(m \log n)$ if GBR is used), where m is the number of edges and n is the number of vertices. The space complexity is determined by the combined size of the generated graphs, which we show to be $O(m+n)$ in Section 9.1.4.

Upper Bound on Total Vertices and Edges

The total number of vertices and edges in the entire series of graphs $G_0, \ldots, G_s$, determines the input size for many of the algorithms in *Nerstrand*. If only a single edge is collapsed between successive graphs such that $n_{i+1} = n_i - 1$ and $m_{i+1} = m_i - 1$, the total number of vertices and edges processed would be $n^2/2$ and $m^2/2$ respectively giving a computational and space complexity of at least $O(m^2 + n^2)$. We address this

issue by stopping coarsening when the rate of contraction slows beyond $|G_i| > \alpha|G_{i-1}|$ where $0 < \alpha < 1.0$. Here $|G|$ represents the size of the graph, this can be in terms of the number of vertices, the number of edges, or a combination of the two. The total number of vertices and edges processed can then be represented as the sum of a geometric series:

$$\sum_{i=0}^{s} |G_i| = \sum_{i=0}^{s} |G_0|\alpha^i = |G_0|\frac{1-\alpha^{s+1}}{1-\alpha}. \tag{9.3}$$

Since a graph must contain at least one vertex (and for our purposes at least one edge) we can place on upper bound on s of $\log_\alpha(1/|G_0|)$. Plugging this in for s in equation (9.3) we get:

$$|G_0|\frac{1-\alpha^{\log_\alpha(\frac{1}{|G_0|})\alpha}}{1-\alpha} = |G_0|\frac{1-\frac{\alpha}{|G_0|}}{1-\alpha} < \frac{|G_0|}{1-\alpha}.$$

Since α is a constant, we can see then that the total number of vertices is $O(n)$ and the total number of edges is $O(m)$. That is, $\sum_{i=0}^{s} n_i = O(n)$ and $\sum_{i=0}^{s} m_i = O(m)$. Our choice of α not only changes the constants involved in these complexities, but also the size of G_s, which affects the quality of the clustering and the amount of computation required during initial clustering.

Coarsening Complexity

In the standard matching aggregation scheme (MAT), each vertex v chooses the unmatched neighbor that maximizes equation (9.1). This requires each vertex to scan through all of its edges, which makes this an $O(m+n)$ operation.

In the two-hop matching aggregation scheme (M2M), we first perform the same operations as MAT ($O(m+n)$). Then, we follow the procedure as detailed in Chapter 6, which has a complexity of $O(m+n)$. Thus, MAT also has a complexity of $O(m+n)$.

In the first choice grouping aggregation scheme (FCG), each vertex v chooses one of its neighbors with which to match. The degree of groupings are updated incrementally as they are formed in $O(1)$ time, which allows determining the degree of a grouping g in $O(1)$ time. As v scans through its edges to determine with whom to match, it sums up the weight of edges connected to each grouping using a hash table, which takes $O(1)$ time per edge. This allows us to look up $d_v(g)$ in $O(1)$. As a result, FCG can be done in $O(m+n)$.

To construct G_{i+1} based on the aggregation of G_i, we iterate over the set of vertices V_i in G_i. When we encounter a vertex $v \in V_i$ that is matched with a vertex $u \in V_i$ with a lower label (or with itself), we construct the new vertex $c \in V_{i+1}$. We merge the adjacency lists of v and u via a hash table using the corresponding coarse vertex numbers as keys. This allows us to combine edges to a vertex $w \in \Gamma(v), \in \Gamma(u)$ as well as edges to vertices x and y that have been aggregated together. This translates to operating on each vertex in the graph and inserting each edge into a hash table which is an $O(1)$ operation, which also gives us a complexity of $O(m+n)$ for contracting a graph with n vertices and m edges. Thus, coarsening G_i to G_{i+1} requires $O(m_i+n_i)$ time, and storing G_{i+1} requires $O(m_{i+1}+n_{i+1})$ space. Since we established that $\sum_{i=0}^{s} n_i = O(n)$ and $\sum_{i=0}^{s} m_i = O(m)$ in Section 9.1.4, we can then say that the coarsening phase takes $O(m+n)$ time.

Initial Clustering Complexity

In order to analyze the complexity in the context of initial clustering, let $n_s = |V_s|$ and $m_s = |E_s|$ represent the number of vertices and number of edges in G_s, respectively. Setting each vertex to be a singleton cluster takes $O(1)$ time per vertex, and thus $O(n_s)$ time total, and $O(n_s)$ space for the cluster labels. Then, performing a pass of Random Boundary Refinement on the n_s clusters takes $O(n_s + m_s)$ time as described in Section 9.1.4. A constant number of clusterings are created, so in total the complexity of initial clustering is $O(n_s + m_s)$. The only bounds on the size of the input graph for initial clustering is $n_s \leq n$ and $m_s \leq m$, thus the complexity of initial clustering is bounded by $O(n + m)$.

Uncoarsening Complexity

Projection is a simple lookup in two arrays for each vertex in the fine graph G_i, thus projection is an $O(n_i)$ operation per graph. Since we know that there are $O(n)$ vertices total in all of the graphs of the multilevel hierarchy, we know that the total complexity of projection is $O(n)$.

In Random Boundary Refinement, the list of boundary vertices can be permuted in $O(n_i)$ time. Each vertex v is visited once per pass, and at most $d(v)$ edges will be inspected when deciding to move v, and at most $d(v)$ clusters will need to be updated if

v is moved. So in the worse case we will need to visit n_i boundary vertices, and we may need to inspect up to m_i edges, and if every vertex is moved then m_i cluster updates will need to be performed. This gives us a complexity of $O(m_i + n_i)$ per pass. By limiting the number of passes that can be performed to a constant number (we found eight to work well), we can see that Random Boundary Refinement takes at most $O(m + n)$ time.

GBR performs the same operations as RBR with the addition of inserting, updating, and extracting vertices from the priority queue, which dominates the runtime. The priority queue contains up to n_i vertices and up to m_i updates can be performed upon it. Which means per graph, refinement takes $O(m_i \log n_i)$ time using a binary heap implementation [151]. And then for all graphs in the multilevel hierarchy we have:

$$O\left(\sum_{i=0}^{s} m_i \log n_i\right) \leq O\left(\left(\sum_{i=0}^{s} m_i\right) \log\left(\sum_{i=0}^{s} n_i\right)\right).$$

We previously established that $\sum_{i=0}^{s} m_i = O(m)$ and $\sum_{i=0}^{s} n_i = O(n)$, so using replacement we can see that the total complexity of GBR is $O(m \log n)$. This however is pessimistic for the objective of modularity, as it tends to favor large clusters as shown by Fortunato and Barthelemy [152], and good clusterings tend to have a set of *core* vertices on the interior of clusters (not part of boundaries) as shown by Ovelgönne and Geyer-Shulz [153].

9.2 Parallel Clustering Methods

In order to allow *Nerstrand* to take advantage of modern compute architectures, we developed shared memory parallel versions of the previously outlined algorithms. We developed methods for assigning vertices to threads in a manner that balances the number of edges for which a thread is responsible. For parallelizing the coarsening phase, we introduce a method for contracting groups of vertices together in an unprotected fashion and resolving broken groupings. Finally, we introduce a method for performing boundary refinement in parallel for the modularity objective.

Our general approach to parallelization follows the coarse-grained model, where threads allocate their own memory and synchronization points are minimized as shown to be effective in Chapter 5. Each thread manages its own subset of vertices of the

original graph. We use the CSR sparse matrix data structure for storing the graph. Each thread allocates its own CSR structure to store its vertices and incident edges. Each thread is responsible for performing the computation associated with its vertices and edges.

9.2.1 Graph Distribution

For distributing vertices and their associated edges among threads, we experimented with three strategies. All three strategies balance the number of edges assigned to each thread, as this is the dominating factor in the runtime. This builds upon the strategies for data distribution in graph partitioning presented in Chapter 5.

The first strategy, which we will refer to as a *block* distribution, preserves the original ordering of the vertices of the graph, and assigns a continuous chunk of vertices to each thread such that the sum of the degrees is roughly $2m/p$, where p is the number of threads. This strategy has the benefit of preserving memory friendly orderings if they exist. However, it can lead to significantly different numbers of vertices being assigned to threads if the vertex degrees are not evenly distributed in the original ordering.

The second strategy, which we will refer to as a *cyclic* distribution, permutes the vertex order in a cyclic fashion using cycles of size p. That is, the array's indices will be reordered to $\{1, p+1, 2p+1, \ldots, 2, p+2, 2p+2, \ldots\}$. Then, each thread is assigned a continuous chunk of permuted vertices such that the sum of the degrees is roughly $2m/p$. Note that this is different from a traditional cyclic distribution in that a thread may be assigned vertices from multiple cycles. This strategy has the benefit of leading to a more even vertex distribution when the vertex degrees of the original ordering are not evenly distributed. However, it sacrifices the benefits of orderings that are memory friendly.

The third strategy, which we will refer to as a *block-cyclic* distribution, attempts to combine the best of both of these strategies. It permutes the vertex order in a block-cyclic fashion. That is, a blocked array of vertices $\{B_1, B_2, B3, \ldots\}$ will be reordered to $\{B_1, B_{p+1}, B_{2p+1}, \ldots, B_2, B_{p+2}, B_{2p+2}, \ldots\}$. Using larger blocks will mean more of the original ordering will be preserved and possibly any memory friendly properties, but will increase the likely hood that the vertices will not be balanced among threads. Using smaller blocks will have the opposite effect. As in the cyclic distribution, each thread

is assigned a continuous chunk of permuted vertices such that the sum of the degrees is roughly $2m/p$.

9.2.2 Coarsening

For parallelizing aggregation, we update the data structures for recording vertex matchings/groupings without using locks or exclusive access patterns, allowing race conditions. We then fix the broken matchings caused by race conditions after attempting to match/-group all vertices using the technique described in Chapter 5.

This however, does not apply to aggregation schemes where more than two vertices can be aggregated together at once, as is the case with FCG. To address this, we developed a parallel method for grouping vertices in an unprotected fashion. We generalize M from being a matching vector to that of a *grouping* vector, where aggregated vertices in M form a cycle of arbitrary length. If a grouping contains the vertices v, u, and w, then $M(v) = u$, $M(u) = w$, and $M(w) = v$.

To accomplish this during aggregation, all vertices are initially grouped with themselves, $M(v) = v$. Then, to add the vertex v to the vertex u's grouping, we set $M(v) = M(u)$, and $M(u) = v$. This means that a valid grouping vector M will contain only cycles.

However, performing updates to this vector without synchronization allows for broken cycles. Because M is initialized to be all length one cycles, and every write to M is a valid vertex number, we know that every index in M is a valid vertex number, and thus a valid index in M. Then, for every vertex v, the linked list created by following the indices $M(v), M(M(v)), \ldots, M(u)$, must be non-terminating, so we know that v must either be part of a cycle, or part of a *tail* connected to a cycle. For architectures in which the writing of words is not atomic (i.e., two threads writing to the same location could result in an invalid vertex number being written), a simple validity check can be added. Vertices that are part of a tail are not part of a valid grouping, and must be cleaned up.

In order to cleanup these tails, the following method described in Algorithm 5 is used which does not require synchronization. Each thread marks all of its vertices as not finalized. Then for each vertex v that a thread owns that is not marked as finalized, the indices in M are followed until a cycle is found using a hash table (line 10). If v

Algorithm 5 Parallel Group Cleanup

```
1:  function GroupCleanup(G(V, E),M)
2:      T ← all vertices owned by this thread
3:      Mark all v ∈ T as not finalized
4:      for all v ∈ T do
5:          if v is not finalized then
6:              H ← a HashTable
7:              u ← v
8:              for i ← 1 to max group size do
9:                  u ← M(v)
10:                 if u ∈ H then
11:                     if u = v then
12:                         if Minimum w ∈ H is owned by this thread then
13:                             Mark all w ∈ H as finalized
14:                         end if
15:                     else
16:                         M(v) ← v and mark v as finalized
17:                     end if
18:                     Break
19:                 else
20:                     Insert u into H
21:                 end if
22:             end for
23:             if v has not been finalized then
24:                 M(v) ← v and mark v as finalized
25:             end if
26:         end if
27:     end for
28: end function
```

is part of that cycle, and v is the owner of the cycle (we use the lowest vertex number in the cycle as the owner), then all vertices in the cycle are marked as finalized. If v

is not part that cycle, it is matched with itself. This leaves us with a valid M vector where every vertex is part of a cycle (including cycles of length one). To avoid creating large cycles, during aggregation the size of groups of vertices are tracked, and vertices are only allowed to join groups smaller than a maximum size (we use 1024).

Contraction is an inherently parallel process, as for any matching or group of vertices being collapsed, the creation of the resulting coarse vertex and coarse edges depends only upon the finer graph G_i and matching/grouping vector M. Threads are responsible for contracting the matchings/groups of fine vertices which form the coarse vertices they will own.

9.2.3 Initial Clustering

For a moderate number of threads, the initial clustering stage lends itself well to parallelization, where each thread creates one or more of the initial clusterings, and a reduction operation is performed at the end to choose the best one. That is, each thread performs refinement on the coarse graph with each vertex initialized as a singleton cluster. The only concern for parallelization here is effectively using the cache hierarchy to reduce the total memory bandwidth required. For large numbers of threads and sufficiently large coarse graphs, each thread initialize the vertices it owns to singleton clusters, and then the threads work cooperatively to create each initial clustering using the parallel formulation of refinement described below in Section 9.2.4.

9.2.4 Uncoarsening

Projection is also an inherently parallel process, as each thread can independently perform cluster projection on the vertices it owns. Conversely, refinement is an inherently serial process.

Because the gains associated with moving a vertex v from the cluster C_i to the cluster C_j depends on the degree information C_i and C_j, we cannot guarantee that moving vertices in parallel will result in a positive net gain in modularity. Having the owning thread lock the pair of clusters C_i and C_j before moving the vertex v would allow us to guarantee we only make positive gain moves, but this would greatly limit the amount of parallelism.

Algorithm 6 Parallel Random Boundary Refinement

```
1: function ParRBR(G(V,E),C)
2:     repeat
3:         C' ← C.
4:         D ← empty List
5:         for all Boundary vertices v this thread owns in random order do
6:             for all c ∈ clusters of Γ(v) do
7:                 if ΔQ(v,c) > ΔQ(v,C'(v)) then
8:                     C'(v) ← c
9:                 end if
10:            end for
11:            if v was moved then
12:                Add move to D and apply local updates to C'
13:            end if
14:        end for
15:        Synchronize threads
16:        C' ← prospective changes from all threads
17:        for all m ∈ D in reverse order do
18:            if ΔQ(v,c) ≤ 0 then
19:                Remove m from D and rollback local updates to C'
20:            end if
21:        end for
22:        Synchronize threads
23:        Apply remote updates to C' and reduce to C
24:        Synchronize threads
25:    until No moves are made or max. # of iterations completed
26:    return C
27: end function
```

Our parallel refinement algorithm is described in Algorithm 6. Instead of using locking clusters, each thread makes a private copy of the global clustering state. This private copy is updated by the thread as it moves the vertices that it owns (line 12).

Because each thread is unaware of the moves being made by other threads, a move that it sees as a positive gain move, may actually result in a loss of modularity.

After all threads have made their desired moves, the global clustering state is updated. Each thread then makes a pass over its selected set of moves, a *roll-back* pass, where it re-evaluates each of its moves in reverse order. This can be see on line 17. If, with the updated cluster information, the move no longer results in a positive gain, the move is rolled back. Note that this does not guarantee that no negative gain moves will be made, as rolling back moves in parallel has the same issue as making the initial moves in parallel. To guarantee no modularity loss, the roll-back pass would need to be repeated until no moves were rolled back, and all remaining moves would have been determined positive gain moves based on up-to-date cluster degrees. We opted to use only a single pass to keep the cost of refinement down as we found it sufficient to prevent the majority of negative gain moves. After all of the threads have rolled back undesirable moves, the global clustering information is updated, and another iteration is started.

The clusters in which the neighbors of v reside affects how the internal and external cluster degrees are effected by moving the vertex v. When performing refinement serially, this is not an issue, as only one vertex moves at a time, and cluster degrees can be updated directly.

Consider the edge $\{v, u\}$ and the incident vertices $v \in C_i$ and $u \in C_j$, each owned by a different thread. Vertex v is moved to C_j and u is moved to C_k concurrently. If the thread that owns v directly updates the cluster degrees, then $2\theta\{v, u\}$ will be added to $d_{int}(C_j)$, when the edge is actually between C_j and C_k. Note that if v and u are owned by the same thread, this is not an issue as v and u will not be moved concurrently.

To solve this problem, we developed a method for handling cluster degree updates that is order independent. Our new method of processing cluster degree updates splits the updates into two distinct parts: *move updates* made by the moving vertex v, and *neighbor updates* made by each neighbor of v. Neighbor updates can be classified as local, where the moving thread also owns the neighbor, and as remote, where the neighbor is owned by a different thread. Move updates and local neighbor updates are applied to the private copies of the cluster degrees as moves are made. Remote neighbor updates are applied afterwards as part of the global clustering state update.

For the move update, the thread that owns the moving vertex v updates its local cluster degrees. For updating the source cluster C_i's internal degree

$$\Delta d_{int}(C_i) = -d_{C_i}(v),$$

C_i's external degree

$$\Delta d_{ext}(C_i) = \frac{d_{ext}(v) - d_{C_i}(v)}{2},$$

the destination cluster C_j's internal degree

$$\Delta d_{int}(C_j) = d_{C_j}(v), \tag{9.4}$$

and C_j's external degree

$$\Delta d_{ext}(C_j) = \frac{d_{ext}(v) + d_{C_i}(v) - d_{C_j}(v)}{2}.$$

For the neighbor update, the thread that owns the adjacent vertex u to the moving vertex v performs updates associated with the edge $\{v, u\}$. The source cluster C_i's internal degree is changed by

$$\Delta d_{int}(C_i) = \begin{cases} -\theta(\{v,u\}) & \text{if } u \in C_i \\ 0 & \text{else} \end{cases},$$

and its external degree is changed by

$$\Delta d_{ext}(C_i) = \begin{cases} \theta(\{v,u\})/2 & \text{if } u \in C_i \\ -\theta(\{v,u\})/2 & \text{else} \end{cases}.$$

The destination cluster C_j's internal degree is changed by

$$\Delta d_{int}(C_j) = \begin{cases} \theta(\{v,u\}) & \text{if } u \in C_j \\ 0 & \text{else} \end{cases}, \tag{9.5}$$

and its external degree is changed by

$$\Delta d_{ext}(C_j) = \begin{cases} -\theta(\{v,u\})/2 & \text{if } u \in C_j \\ \theta(\{v,u\})/2 & \text{else} \end{cases}.$$

By splitting the updates like this, they can be applied independent of the order in which the vertices were moved.

Applying this to our previous example where the vertex v is moved to C_j and the vertex u is moved to C_k concurrently, these order independent updates result in the correct cluster degree changes. First, $\theta\{v,u\}$ would get added to $d_{int}(C_j)$ as part of the move update via equation (9.4), and then the neighbor update performed after u has moved to C_k then removes $\theta\{v,u\}$ from $d_{int}(C_j)$ via equation (9.5). This has the correct net effect of leaving $d_{int}(C_j)$ unchanged with respect to the edge $\{v,u\}$.

For Random Boundary Refinement, each thread visits the boundary vertices it owns in random order. To visit vertices in order of their potential gain for performing Greedy Boundary Refinement in parallel, each thread maintains a priority queue containing the boundary vertices which it owns.

9.2.5 Parallel Complexity

The overall complexity for the parallel algorithms in *Nerstrand* is the sum of its three phases:

1. Coarsening, $O(m/p + n/p)$.
2. Initial clustering, $O(m/p + n/p + k)$.
3. Uncoarsening, $O(m/p + n/p + k)$ for RBR and $O((m/p)\log(n/p) + k)$ for GBR.

Adding these we get an overall parallel complexity of $O(m/p+n/p+k)$ (and $O((m/p)\log(n/p)+k)$ if GBR is used), where m is the number of edges, n is the number of vertices, p is the number of threads, and k is the number of clusters.

Coarsening Complexity

When using MAT or M2M to coarsen a graph in parallel, each thread is responsible for finding matches for its set of vertices, which entails scanning all incident edges, which makes finding matches for all vertices an $O(m/p + n/p)$ operation. Then, to fix the broken matchings, each thread makes a second cleanup pass over its vertices, which is an $O(n/p)$ operation.

For FCG, it takes $O(m/p + n/p)$ time for threads to group their sets of vertices. Performing the group cleanup requires each thread to iterate over each of its vertices. Then, for each vertex, the inner loop on line 8 of Algorithm 5 can search up to the

maximum group size number of vertices in the worst case. This gives group cleanup an upper bound on runtime of $O(n/p)$, albeit with a very large constant in front (the maximum group size). Even when many of the groupings reach maximum size, this rarely plays a significant factor in the total runtime as the edges in a graph usually greatly outnumber the vertices, and once the owning thread makes a pass over the group, the vertices will get marked as finalized and will not be traversed again.

Contraction requires each thread to iterate over the fine vertices and edges that form the coarse vertices and edges it will own in the next graph. This gives contraction a parallel time complexity of $O(m/p + n/p)$.

Putting all of the parallel complexities from coarsening together, we that parallel complexity of coarsening is $O(m/p + n/p) + O(n/p) = O(m/p + n/p)$.

Initial Clustering Complexity

For small numbers of threads and small coarsest graphs, each thread independently generates and initial clustering in $O(m+n)$ time. However, for large numbers of threads or large coarsest graphs, the initial clusterings are generated cooperatively. Which means that each thread initializes its own n/p vertices to singleton clusters. Then parallel RBR is applied to the singleton clusters, which is shown in the next section to have a parallel complexity of $O(m/p + n/p + k)$ for m edges and n vertices.

Uncoarsening Complexity

Projection is an inherently parallel process where each thread projects the clusters from the vertices in the coarse graph to the n/p fine vertices it owns. This results in $O(n/p)$ time.

For parallel RBR, each thread iterates over its own at most n/p boundary vertices and considers them for moving as in serial RBR. Using the $O(m + n)$ result for serial RBR in Section 9.1.4 for these n/p vertices with m/p incident edges, we then know that the movement pass of parallel RBR takes $O(m/p + n/p)$ time. The additional roll-back pass performs at most the same operations, and thus is also bounded by $O(m/p + n/p)$ time.

Then in the two steps where the state of the clusters is updated, at lines 16 and 23 in Algorithm 6, all of the p states for each of the k clusters must be combined. This

can be done by assigning k/p clusters to each thread and then having the assigned thread combine the p values for each of the clusters it is assigned. This results in a time complexity of $O(pk/p) = O(k)$. Thus parallel RBR takes $O(m/p + n/p + k)$ time. At the coarser levels where k is close to n, we can see that it dominates the runtime. Then, at the finer levels where k is much smaller than n, the $O(m/p + n/p)$ term dominates the runtime.

For parallel GBR, each thread has the extra work of maintaining its priority queue which could contain up to n/p vertices, and make up to m/p updates per iteration. This plus the $O(m/p+n/p)$ for the rollback pass, and $O(k)$ for the cluster state updates makes the parallel complexity for GBR $O((m/p)\log(n/p) + k)$.

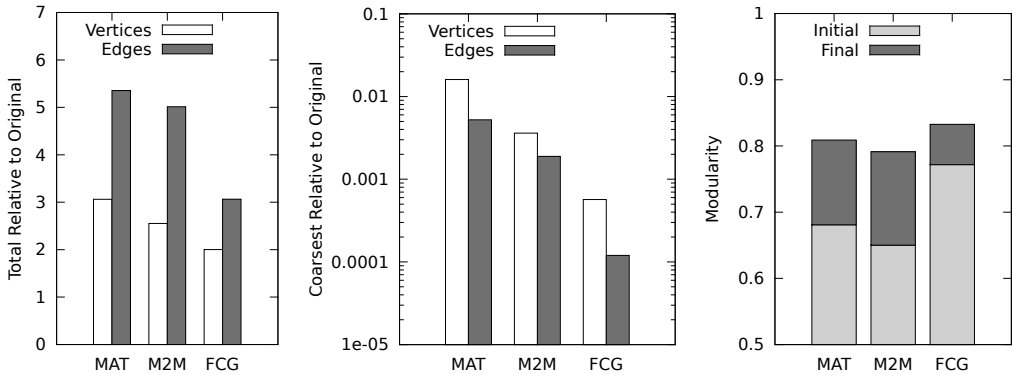
Putting these results together with the $O(n/p)$ time for projection, we can see that parallel uncoarsening takes $O(m/p + n/p + k)$ time for RBR, and $O((m/p)\log(n/p))$ time for GBR.

9.3 Experimental Methodology

The experiments that follow were run on an HP ProLiant BL280c G6 with 2x 8-core Xeon E5-2670 @ 2.6 GHz system with 256GB of memory. We used GCC 4.7 and the accompanying libgomp that conforms to the OpenMP 3.1 specification. Unless otherwise noted, all runs were repeated 25 times with different random seeds to get the geometric mean, minimum, or maximum time and modularity.

The serial and parallel algorithms presented in the previous sections are implemented in *Nerstrand*, available at `http://cs.umn.edu/~lasalle/nerstrand`. When run with a single thread, a separate set of functions implementing the serial algorithms are executed. For simplicity, we will refer to single threaded executions of *Nerstrand* as *s-Nerstrand*, and the multithreaded executions as *mt-Nerstrand*.

We compare *s-Nerstrand* against what is currently the fastest [137] available serial method for modularity maximization on large graphs, *Louvain* [136]. We used version 0.2 which is available from `https://sites.google.com/site/findcommunities/`. Because of *Nerstrand*'s similarity with multilevel graph partitioners, we compare against *Metis* [92] using version 5.1.0, available from `http://cs.umn.edu/~metis`. To facilitate finding clusters, we allowed for up to a 50000% imbalance and for the number of



(a) (b) (c)

Figure 9.1: The total number of edges and vertices generated during the multilevel process relative to that of the input graph (a), the size of the coarsest graph relative to the input graph (b), and the mean modularity (c), for each coarsening scheme.

partitions we used powers of two from eight to 16,384, and selected the clustering/partitioning that resulted in the highest modularity.

We also compare *mt-Nerstrand* against the parallel clustering tool *community-el* [138] using version 0.7, available at `http://www.cc.gatech.edu/~jriedy/community-detection/`, and the implementations of parallel label propagation (*PLP*) and the parallel louvain method with refinement (*PLMR*) provided by NetworKit [140] using version 3.1 available at `https://networkit.iti.kit.edu/`.

9.4 Serial Results

Sections 9.4.1 through 9.4.3 present the results of our experiments designed to evaluate the different schemes for coarsening, initial clustering, and uncoarsening.

In Section 9.4.4 we present the best of these schemes as implemented in *s-Nerstrand* compared against the *Louvain* method. This comparison is in terms of clustering quality as well as runtime performance.

9.4.1 Aggregation Schemes

We evaluated the three aggregation schemes (MAT, M2M, and FCG) using three criteria: rate of contraction, size of coarsest graph, and effect on modularity.

Rate of Contraction

We measured the rate of contraction by using the total number of vertices and edges found in G_0 through G_s, as this directly correlates to the amount of work done in the coarsening and uncoarsening phases, and the total amount of space used. This is shown in Figure 9.1(a). The plain vertex matching scheme, MAT, did the worst, on average generating a total of 3.1 times as many vertices and 5.4 times as many edges as in the original graph. M2M did better, generating a total of 2.6 times as many vertices and 5.0 times as many edges as in the original graph. This improvement is the result of a more complete matching made possible by two-hop matches. Notice however that this primarily resulted in fewer vertices being generated, and only marginally decreased the number of edges generated. This is because when a two-hop match is made, edges are only combined, not collapsed. FCG did the best, generating only 1.7 times as many vertices and 2.8 times as many edges as in the original graph. This is because more than two vertices are aggregated together at a time, greatly reducing the number of vertices in coarser graphs, and increasing the number of edges that get combined. In addition to this, because we are targeting groups of highly connected vertices for collapsing, we contract a large number of edges with each coarse graph generated. This shows that while using the 0.95 minimum coarsening rate allows for up to 20 times as much space required as the size of the input graph as shown in Section 9.1.4, in practice for FCG it is closer to only 3 times as much space as required by the original graph.

Size of Coarsest Graph

Figure 9.1(b) shows the number of vertices/edges in the coarsest graph divided by the number of vertices/edges in the original graph (y-axis is in log-scale). As expected, M2M outperformed MAT generating a coarsest graph of roughly half the size on average, a result of additional vertices being matched with two-hop neighbors. FCG greatly outperformed the other methods averaging a coarsest graph with an order of magnitude

less vertices and two orders of magnitudes less edges. Notice that FCG significantly reduced the density of the final graph. This is because as FCG merges groups of vertices together, these groups are supposed to represent clusters, which by definition should have a large number of internal edges, and few external edges.

Effect on Modularity

Figure 9.1(c) shows the modularity after the initial clustering of the coarsest graph, as well as the final modularity of the clustering refined and applied to the original graph. At the initial clustering phase, M2M did the worst, with an average modularity of 0.650, followed by MAT at 0.681. This difference in modularity between the two matching schemes can be attributed to the gain agnostic two-hop matches allowed by M2M. FCG did the best, with an average modularity 0.771. This is because where the two matching schemes will only choose the maximum gain matching from unmatched neighbors, FCG selects the maximum gain matching/grouping from among all neighbors. After refinement, MAT and M2M were much closer, averaging 0.809 and 0.791 respectively. The reason for M2M closing the modularity gap, is that in refinement, many of the negative gain two-hop matches are undone. Just as FCG did the best at creating a sparse coarse graph, it also resulted in the highest average modularity, of 0.832. This is again due to FCG always choosing the highest gain merges.

Due to the success of FCG in both reducing the size of the graph and in terms of modularity, we elected to use it as the coarsening scheme in *Nerstrand*.

9.4.2 Initial Clustering Schemes

A good initial clustering scheme will have relatively stable runtime and solution quality over a variety of inputs. To better observe their robustness, we evaluated the initial clustering scheme in the context of all three coarsening schemes (MAT, M2M, and FCG).

We evaluated generating the initial clustering by initializing each vertex to its own cluster and refining it using both Greedy Boundary Refinement (VTX-GBR) as well as using Random Boundary Refinement (VTX-RBR). We ran VTX-RBR with 16 different random seeds to generate different initial clusterings and selected the best one. We compared these two approaches to that of a modified version of the algorithm by Clauset et al. [133] that takes into account the weight of collapsed edges (CNM).

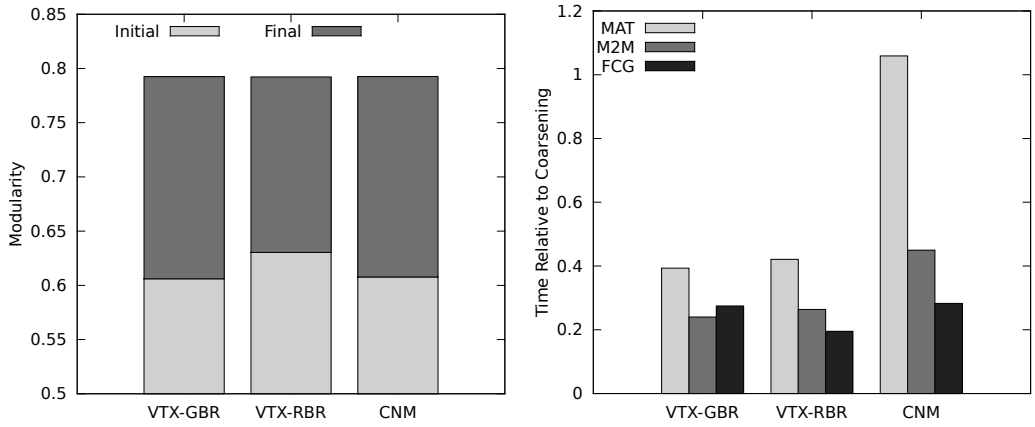


Figure 9.2: The mean modularity (a) and the mean runtime relative to coarsening (b) of the initial clustering schemes.

Figure 9.2(a) shows the quality of the initial clustering solutions both before and after refinement. The VTX-RBR method generated clusterings with the highest modularity at the end of initial clustering, 0.630. The VTX-GBR and CNM algorithms were similar in performance with modularities of 0.606 and 0.608 respectively. However, once these clusterings were projected and refined to the original graphs, all three schemes had a final modularity of 0.792. This show the power of the multilevel paradigm, where a large amount of the solution quality is a result of coarsening, which was performed identically for all three schemes, and in refinement where rough solutions can often be significantly improved.

Figure 9.2(b) shows the amount of time spent in initial clustering and uncoarsening for each coarsening scheme relative to the amount of time spent in coarsening. The fastest overall initial clustering scheme was VTX-RBR, taking 27.9% of the time of coarsening. Only slightly slower, was VTX-GBR, taking 29.6% of the time of coarsening. VTX-RBR managed to be faster than VTX-GBR even though it made 16 clusterings. This is due to the different complexities of vertex traversal: random permutation versus a priority queue. While on finer graphs GBR exhibits near linear runtimes as only a fraction of the vertices are on the boundary, VTX-GBR starts with all vertices on the boundary, thus VTX-GBR performs very close to its worse case runtime of $O(m \log n)$.

The CNM algorithm was the slowest, taking 51.3% of the time of coarsening.

Based on these findings, we selected VTX-RBR as the initial clustering scheme for use in *Nerstrand*.

9.4.3 Refinement Schemes

Table 9.1: Comparison of Refinement Schemes.

Method	Mod. Improvement	Runtime (s)
RBR	0.01705	3.62906
GBR	0.01708	8.08687

The effect of the two different refinement schemes (RBR and GBR) on modularity as well as their runtimes are shown in Table 9.1. Their modularity improvement was nearly identical, with GBR improving modularity only 0.15% more than RBR. Although the order in which vertices were visited during refinement appears to not impact the modularity improvement, it does however affect the number of refinement passes required at each level. GBR on average made 1.07 passes before reaching a steady state, whereas RBR made an average of 2.10 passes before reaching a steady state. The cost of maintaining the priority queue caused GBR to be significantly slower, taking 2.23 times longer than RBR. This is a product of the $O(\log n)$ time required to insert, update, and remove vertices from the priority in GBR, as opposed to the randomly permuted list used in RBR in which vertices are only inserted in $O(1)$ time.

Given that both schemes improve the quality of clusterings nearly the same amount, we opted to use RBR as the refinement scheme in *Nerstrand* due to its lower runtime.

9.4.4 Performance

Table 9.2 shows the mean and maximum modularity and the mean runtime for *s-Nerstrand* creating clusterings of the graphs from the 10th DIMACS Implementation Challenge [48]. For comparison, the modularity and the runtime of the best clusterings (for which runtime was reported) from the challenge are shown on the right. While taking several orders of magnitude less time, *s-Nerstrand* finds clusterings with modularities equal to, or within a few percentage points of the best clusterings found in the

Table 9.2: Comparison of *s-Nerstrand* against the best results achieved in the DIMACS challenge on graph clustering.

	s-Nerstrand			DIMACS Best	
Graph	Mean	Max.	Time (s)	Max.	Time (s)
333SP	0.983	0.984	3.891	0.989	58614.5
G_n_pin_pout	0.480	0.485	1.105	0.500	3887.5
PGPgiantcompo	0.879	0.882	0.009	0.887	114.7
as-22july06	0.670	0.672	0.021	0.678	395.2
astro-ph	0.731	0.734	0.031	0.744	714.5
audikw1	0.913	0.914	3.484	0.917	75872.8
belgium.osm	0.993	0.993	0.771	0.995	6175.8
cage15	0.884	0.890	19.127	0.903	157390.2
caidaRouterLeve.	0.864	0.865	0.223	0.872	4859.2
celegans_metab.	0.442	0.446	0.001	0.452	4.4
citationCitesee.	0.817	0.819	0.467	0.824	4658.0
coAuthorsCitese.	0.895	0.896	0.327	0.905	5477.8
coPapersDBLP	0.855	0.857	2.024	0.867	36197.3
cond-mat-2005	0.729	0.733	0.066	0.746	2456.9
email	0.572	0.578	0.002	0.582	8.9
eu-2005	0.939	0.940	2.149	0.942	20488.6
in-2004	0.980	0.981	1.766	0.981	14639.0
ldoor	0.964	0.965	1.943	0.969	29138.2
luxembourg.osm	0.984	0.985	0.050	0.989	2453.6
memplus	0.689	0.694	0.020	0.700	193.6
polblogs	0.426	0.427	0.002	0.427	6.7
power	0.937	0.938	0.003	0.940	16.7
preferentialAtt.	0.276	0.278	1.241	0.302	81183.1
rgg_n_2_17_.	0.972	0.973	0.129	0.978	2251.7
smallworld	0.770	0.771	0.300	0.793	1007.8
uk-2002	0.990	0.990	45.028	0.990	478859.9

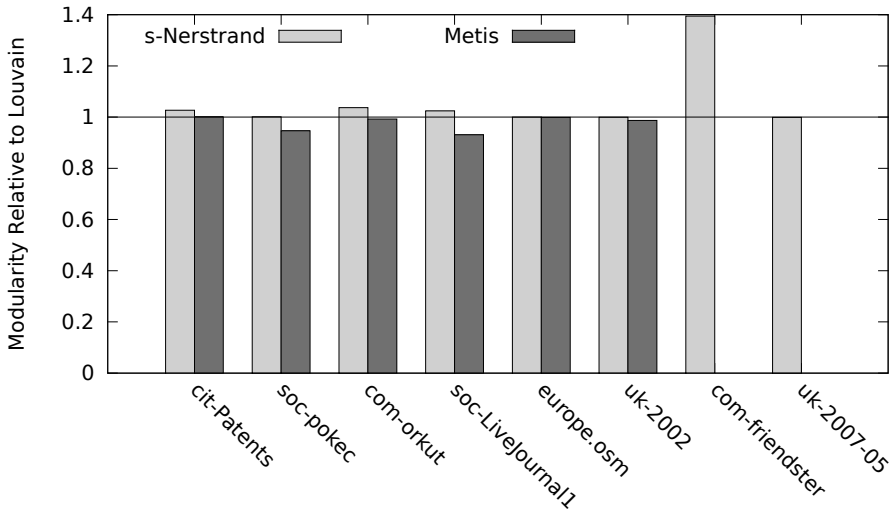


Figure 9.3: The modularity of clusterings generated by *s-Nerstrand* relative to *Louvain*.

challenge.

The high mean and maximum modularity of the clusterings generated by *s-Nerstrand* in 25 runs demonstrates the effectiveness of the multilevel paradigm for maximizing modularity. By only aggregating vertices together for which the associated modularity gain is positive, we increase the lower bound on the quality of clusterings that can be generated at the coarsest levels. Because the coarse graph is of small size, we can afford to make many initial clusterings of it, and choose the best one, giving us a further guarantee on the quality of the clustering being generated. Finally, during during refinement we are able to continue to increase the modularity of the clustering at each level.

The quality of the clusterings generated by *s-Nerstrand* and *Metis* relative to *Louvain* are shown in Figure 9.3. In terms of clustering quality, *s-Nerstrand* generated clusterings with an average modularity equal to or slightly greater than *Louvain*. Across all eight graphs, *s-Nerstrand* produced clusterings that were on average 5.3% better. Although *s-Nerstrand* and *Louvain* produced clusterings of nearly identical modularity (differing by less than 0.1%) on `soc-pokec`, `europe.osm`, `uk-2002`, and `uk-2007-05`, *s-Nerstrand* produced clusterings with noticeably higher modularities for `cit-Patents`, `com-orkut`, `soc-LiveJournal1`, and `com-friendster`, at 2.7%, 3.7%, 2.4%, and 39.4% respectively.

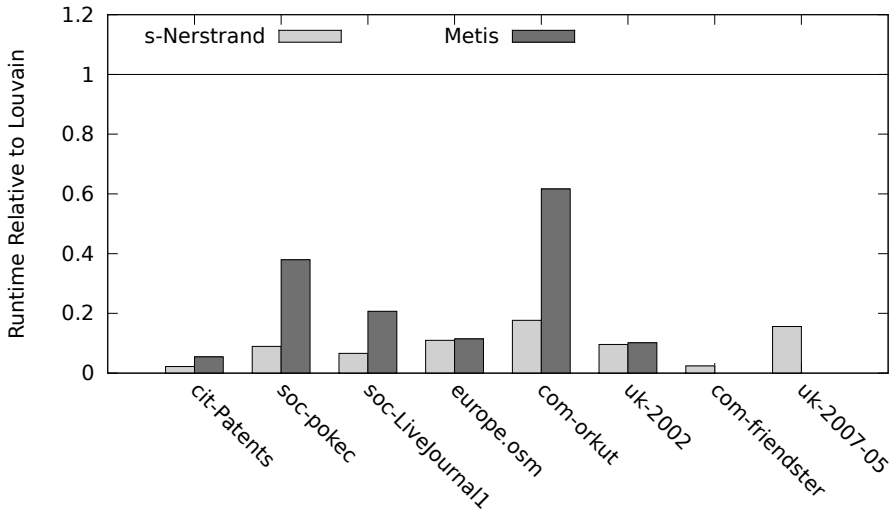


Figure 9.4: The runtime of *s-Nerstrand* relative to *Louvain*.

The high quality of clusterings being generated by *s-Nerstrand* despite its aggregation approach using only a single pass, is the result of the refinement performed on each of the coarse graphs. The significantly higher modularity of clusterings found by *s-Nerstrand* for `com-friendster`, is the result of *s-Nerstrand* being able to contract the graph down to ten vertices, whereas *Louvain* stopped at over 50 thousand and produced a much larger number of communities.

Due to its slower rate of contraction, *Metis* was unable to cluster the two largest graphs in the 256GB of memory in our test machine. *Metis* is able to produce clusterings with modularities that are within 1–2% of *s-Nerstrand* for graphs with strong community structure (`europe.osm` and `uk-2002`), the edgecut and modularity objectives both find areas of extremely low connectivity to place cluster boundaries. However, for graphs with less strong community structures (`cit-Patents`, `soc-pokec`, `com-orkut`, and `soc-LiveJournal1`), *Metis* produces clusterings that are 3–10% lower than *s-Nerstrand* as the two objectives diverge. In addition to this, the number of clusters must be known a priori for the algorithms in *Metis*.

The runtimes for generating clusterings for *s-Nerstrand* and *Metis* relative to *Louvain* are shown in Figure 9.4. *s-Nerstrand* outperformed *Metis* and *Louvain* for all graphs in this experiment in terms of computation time: 1.04–4.25 times faster than

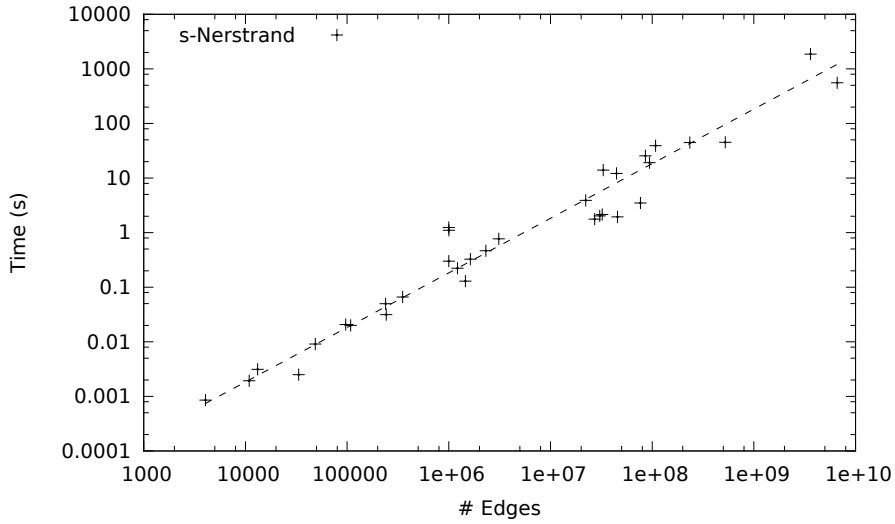


Figure 9.5: The scaling of *s-Nerstrand* with respect to the number of edges in a graph.

Metis and 5.66–44.9 times faster than *Louvain*. The lower runtime of *s-Nerstrand* than *Metis* is the result of its superior contraction rate made possible by FCG aggregation that groups many vertices together at a time while decreasing the edge density in resulting graphs. This difference in runtime between *s-Nerstrand* and *Louvain* can be attributed to the different ways in which aggregation is performed. In *s-Nerstrand*, each vertex is processed only once, whereas *Louvain* repeatedly processes its vertices until a local maxima in modularity is found.

The scaling of *s-Nerstrand* with respect to the number of edges in the input graph is shown in Figure 9.5, with 25 graphs from the DIMACS Challenge [48] (shown in Table 9.2 in addition to the six used previous and the two large graphs: `com-friendster` and `uk-2007-05`. A line has been fitted to these point to show their trend, with a slope of 183 nanoseconds per edge (or 55 million edges per second). This shows for real world datasets *s-Nerstrand* demonstrates linear scalability with a very small constant factor.

9.5 Parallel Results

In this section we present the results of our experiments for *mt-Nerstrand*. We show that not only does it achieve significant speedup over *s-Nerstrand* and outperforms other

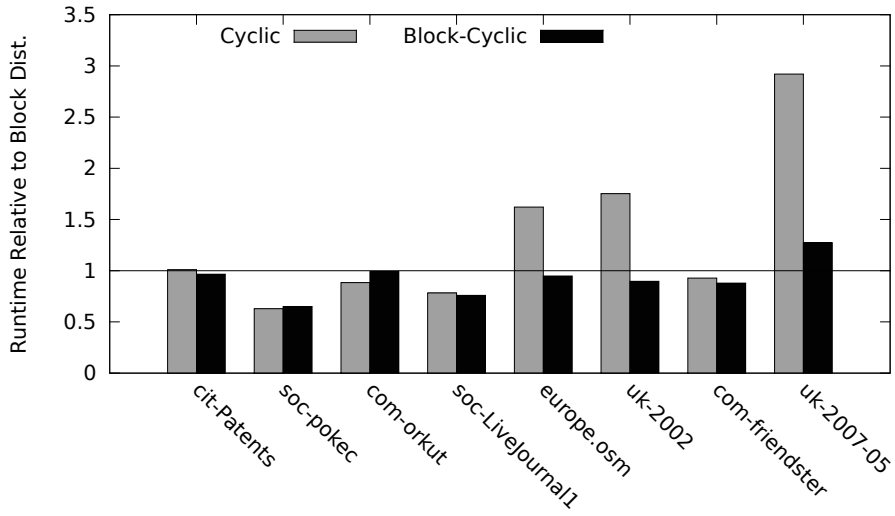


Figure 9.6: The runtimes of each distribution using 16 threads relative to a block distribution.

methods, but does so without making sacrifices in terms of quality.

9.5.1 Graph Distribution

The effects on runtime of the different graph distribution strategies is shown in Figure 9.6. The block-cyclic distribution was run with a block size of $4,096$. Concerning the performance difference between a block distribution and a cyclic distribution, we see an even split where the block distribution performs better for half of the graphs and the cyclic distribution performs better for the other half.

Overall, the block-cyclic distribution performed the best, being the fastest distribution on five of the eight graphs, and on the three graphs where it was not the fastest, it was second, showing its robustness as a distribution strategy. This is because it combines the memory friendly ordering properties of the block distribution and the load balancing properties of the cyclic distribution. For the case where the block-cyclic distribution performed worse than the block distribution, `uk-2007-05`, block-cyclic was as fast or faster in all steps except the most memory intensive step, contraction, where it was just over twice as slow. However, if we increase the block size from $4,096$ to $16,384$, the block-cyclic distribution becomes faster than the block distribution on this graph.

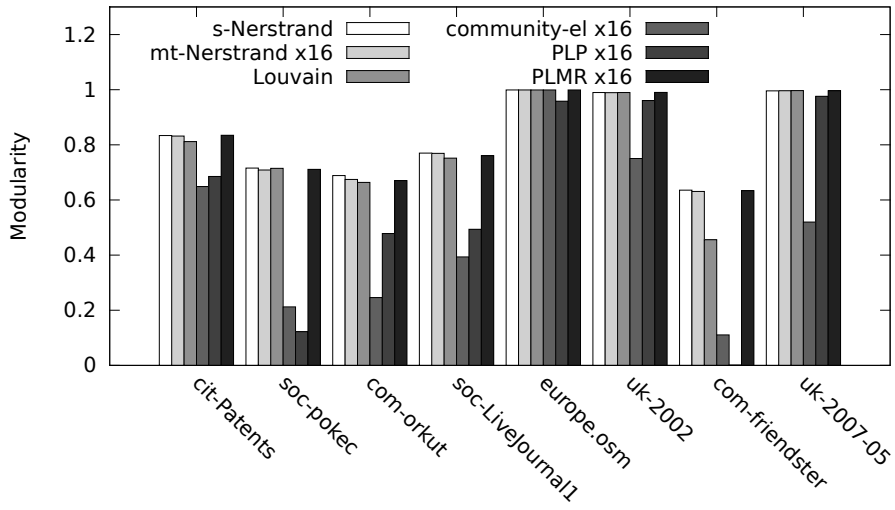


Figure 9.7: The modularity of generated clusterings of the different algorithms.

Where all three distribution schemes balance the number of edges across the threads, the ratio of the maximum number of vertices to the average number assigned to a thread, the vertex imbalance, was highest for the block distribution. The block distribution averaged a vertex imbalance of 4.60 for the eight graphs, and was highest on `uk-2007-05` at 9.12. The cyclic and block-cyclic distributions both averaged vertex imbalances of 1.36, and also had their highest vertex imbalances on `uk-2007-05` at 1.61 and 1.63 respectively.

Due to the block-cyclic distribution's superior performance overall, it is the distribution used by *Nerstrand* in the experiments that follow (continuing to use a block-size of $4,096$).

9.5.2 Quality

The effect on modularity of the parallelizing the serial algorithms in *s-Nerstrand* for *mt-Nerstrand* can be seen in Figure 9.7. We have included the results from *Louvain*, *community-el*, *PLP*, and *PLMR* for comparison. When run with 16 threads, *mt-Nerstrand* shows only minor degradation in cluster quality compared to its serial counter part, averaging 99.5% the modularity of *s-Nerstrand*. This is 4.8% higher modularity than clusterings produced by *Louvain*. Compared to other parallel methods

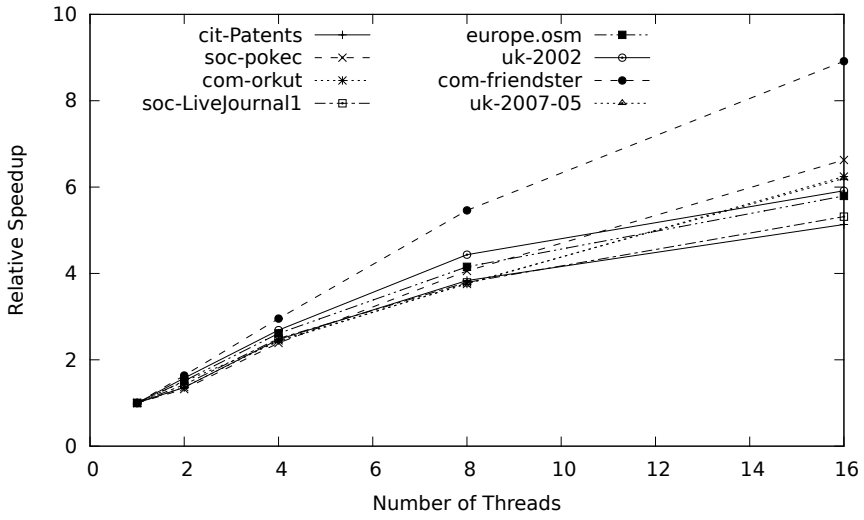


Figure 9.8: The parallel scaling of *mt-Nerstrand* for the eight test graphs.

using 16 threads, *mt-Nerstrand* produced clusterings with 89% higher modularity than *community-el*, and 215% higher than those produced by *PLP*. The clusterings produced by *PLMR* were of near identical modularity to *mt-Nerstrand*, with *mt-Nerstrand* producing clusterings of only 0.07% higher modularity.

The reason *mt-Nerstrand* is able to produce clusterings with modularity similar to that of *s-Nerstrand* is that the quality of the coarsening and initial clustering phases is unaffected by the number of threads. It is not until the refinement step that we see a difference. This is the result of moves being made with partially stale cluster states. However, our results show that this has an extremely small effect on the quality.

The low quality of the clusterings produced by *PLP*, particularly on the social network graphs which tend to have higher inter-cluster connectivity, is due to it not directly optimizing modularity. However, on the web graphs and the citation network where clusters have low inter-cluster connectivity, it was able to find clusterings of modularity within a few percentage points of those found by *mt-Nerstrand*.

9.5.3 Scaling

The speedups achieved by *mt-Nerstrand* with respect to *s-Nerstrand* are shown in Figure 9.8. The mean speedup for all eight graphs using 16 threads was 6.2×. The highest

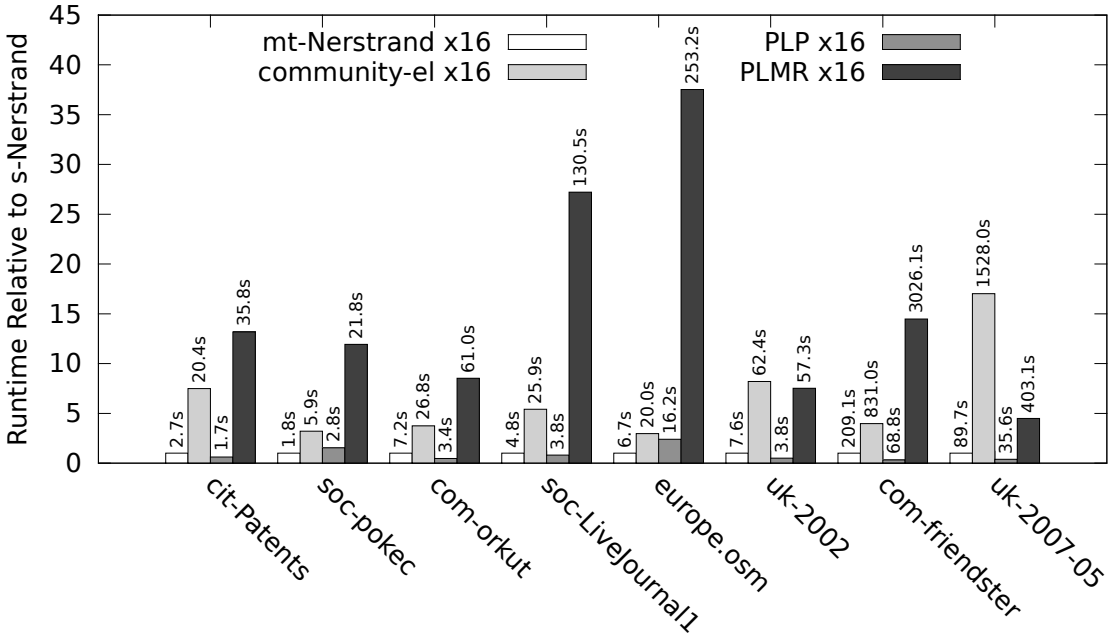


Figure 9.9: The runtimes of the parallel clustering methods relative to *mt-Nerstrand*, with their absolute runtimes listed above.

achieved speedup was 8.91× on the largest social network graph, `com-friendster`, and the lowest speedup of 5.15× was on the patent citation network, `cit-Patents`, which is also the smallest graph used. We did not see as high of a speedup on this graph as a result of refinement performing extra work when done in parallel. For this graph, over twice as many refinement passes were made when using 16 threads as compared to when run serially.

The k component of the $O(m/p + n/p + k)$ parallel complexity of *mt-Nerstrand* played relatively little role in the scaling, as its largest value was for the `uk-2007-05`, at 760 thousand, far below the 3.3 billion edges and 105 million vertices. The smallest value for k was on `com-friendster`, at nine.

Figure 9.9 shows the runtimes of *mt-Nerstrand*, *community-el*[2], *PLP*, and *PLMR*, using 16 threads. The runtime relative to *mt-Nerstrand* is represented by the height of

[2] Timings for *community-el* on the `uk-2002` and `uk-2007-05` graphs were performed with its *coverage* option to terminate the runs early (set to 75% and 50% respectively).

each bar, while the absolute runtime in seconds is displayed at the top of each bar.

The only method faster than *mt-Nerstrand* was *PLP*. Despite the added overheads of using the multilevel paradigm which allows it to find clusterings of significantly higher modularity, *mt-Nerstrand* is only on average 42% slower than *PLP* using 16 threads, up to 204% slower for `com-friendster`, and for europe.osm *mt-Nerstrand* was 142% faster. This high variability in runtime relative to the size of the graph for *PLP* is due the number of iterations it takes for the labels to fully propagate. For `com-friendster` it took seven iterations on average to find a clustering solution, whereas for `europe.osm` it took 546 iterations on average to find a clustering solution. This expounds one of the strengths of the multilevel paradigm. Where label propagation takes $O(\delta)$ iterations to propagate through a cluster with a diameter of δ, the vertex contraction of *mt-Nerstrand* takes only $O(\log(\delta))$ levels to fully contract the cluster as groups of vertices are recursively merged together.

The high parallel performance of *mt-Nerstrand* comes from being based on the already fast algorithms of *s-Nerstrand*. During coarsening, one the most time intensive steps of the multilevel paradigm, *mt-Nerstrand* is able to use the same algorithm as *s-Nerstrand*, and scales well due to the unprotected grouping introduced in Section 9.2.2. The initial coarsening phase of *s-Nerstrand* is inherently parallel, and scales well when all of the threads can fit their data into the cache. Our parallel formulation of boundary refinement with the order-independent updates described in Section 9.2.4, allows us to achieve high modularity in a scalably parallel fashion.

Chapter 10

Conclusion

Multilevel graph methods are complex, with several different sub-processes involving highly irregular access patterns. Achieving high performance on modern parallel architectures over a variety of inputs is a significant challenge. In this thesis we have presented strategies and algorithms for shared memory parallel architectures for graph partitioning, sparse matrix ordering, and graph clustering.

In Chapter 5 we explored the design space of multithreaded graph partitioning and demonstrated the performance improvement it can offer over traditional MPI codes on multicore/multi-processor machines. Our final implementation, *mt-Metis*, is on average over twice as fast as *ParMetis* and *Scotch*. This speedup is due to a combination of avoiding message passing overheads and modifying the existing parallel algorithms used in *ParMetis*. Specifically, our unprotected matching scheme significantly reduces the runtime of the most time consuming phase of multilevel graph partitioning. Beyond the improved speedup, *mt-Metis* also uses significantly less total memory than either *ParMetis* or *Scotch*. This reduced memory footprint plays an important role in enabling the partitioning of large graphs on modern machines which have a decreasing memory to processing element ratio.

In Chapter 6 we presented algorithmic modifications to our multithreaded graph partitioning framework. These modifications resulted in performance increases of 1.5–11.7× and increased strong scaling by 82%, while preserving partition quality. Our modifications include an efficient method for performing two-hop matchings, a new parallel formulation of initial partitioning, a method for reducing communication during

uncoarsening, and implementation level optimizations for coarsening.

In Chapter 7 we presented the Hill-Scanning algorithm, a novel shared memory parallel refinement method for graph partitioning. Our parallel algorithm has the ability to hill climb (break out of local minima), allowing it to find solutions of high quality. By identifying the groups of vertices that form hills before they are moved, we are able to move the group of vertices to the partition of maximum gain. We showed that our method when run serially is competitive with other serial methods, on average 3.5% faster and produces partitionings of equal or greater quality. Unlike other hill-climbing refinement algorithms, the Hill-Scanning algorithm is parallel. We showed that the Hill-Scanning algorithm runs in $O(kn/p + (m/p)\log n)$ time, where k is the number of partitions, n is the number of vertices, m is the number of edges and p is the number of threads. Our strong scaling experiments showed that Hill-Scanning achieves 5.7–16.7× speedup when run with 24 threads, while only producing 0.52% higher edgecuts than when run serially.

In Chapter 8 we presented new shared memory parallel methods for producing minimal balanced vertex separators and fill reducing orderings of sparse matrices. Specifically, we introduced a new parallel refinement scheme for vertex separators that can break out of local minima. We also introduced a task scheduling scheme specifically designed for the nested dissection problem that outperforms OpenMP task schedulers by 40.8%. We implemented these algorithms in *mt-ND-Metis*, and showed that using 16 threads it produces orderings 1.5× faster than *ParMetis* [20] and *Scotch* [21], and 10.1× faster than *ND-Metis* [20]. The orderings produced by *mt-ND-Metis* result in only 1.0% more fill-in and require only 0.7% more operations than those of *ND-Metis* for Cholesky factorization.

In Chapter 9 we presented several approaches to solving the issues associated with adapting the multilevel paradigm for maximizing modularity in serial and in parallel. We adapted the FirstChoice aggregation scheme from graph partitioning to graph clustering such that it is able to effectively maximize modularity. We showed that this aggregation scheme works well for the modularity objective both in terms of quality and in terms of speed. We introduced a robust and fast method for generating clusterings of a contracted graph. We followed these with a modified version of boundary refinement for the modularity objective. We showed the combined computational complexity of these

algorithms is $O(m + n)$. We then presented shared memory parallel versions of these algorithms. This included a means of performing group-based aggregation effectively in parallel, and introducing an order independent method for updating cluster information during refinement without the use exclusive locks. We showed that these shared memory parallel algorithms have parallel complexity of $O(m/p+n/p+k)$, and achieve speedups of 5.1–8.9× over their serial counterparts. We presented these solutions in the form of the multithreaded graph clustering tool *Nerstrand*, which is capable of producing high quality clusterings of large graphs extremely fast. We evaluated this tool on graph with millions vertices and billions of edges. Our tool finds clusterings of equal or better modularity than current methods. *Nerstrand* is fast, finding these clusterings 4.5–27.2× faster than competing methods that produce results of similar quality.

Future work includes extending these algorithms to the distributed setting, where they can take on a hybrid form of shared memory parallelism within a compute node, and distributed memory parallelism across the system. This has the potential to reduce the amount of communication on a system by an order magnitude as well as drastically reduce the memory required. These are both major concerns as we approach ExaScale compute systems [73]. Effective methods for graph partitioning and nested dissection are key to getting performance out of scientific applications on these large systems. As the world of Big Data continues to mature, methods for efficiently clustering massive graphs will become increasingly important.

References

[1] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[2] Yifan Hu. Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[3] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to parallel computing*. Pearson Education, 2003.

[4] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[5] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*. SIAM, 2007.

[6] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *Journal of Experimental Algorithmics (JEA)*, 13:5, 2009.

[7] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning. In *Experimental algorithms*, pages 376–387. Springer, 2011.

[8] Tobias Polzin and Siavash Vahdati Daneshmand. Practical partitioning-based methods for the steiner problem. In *Experimental Algorithms*, pages 241–252. Springer, 2006.

[9] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.

[10] Sabih H Gerez. *Algorithms for VLSI design automation*, volume 8. Wiley New York, 1999.

[11] Timothy A Davis. *Direct methods for sparse linear systems*, volume 2. Siam, 2006.

[12] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[13] Lei Tang and Huan Liu. Community detection and mining in social media. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 2(1):1–137, 2010.

[14] Xujuan Zhou, Yue Xu, Yuefeng Li, Audun Josang, and Clive Cox. The state-of-the-art in personalized recommender systems for social networking. *Artificial Intelligence Review*, 37(2):119–132, 2012.

[15] Miao Xu, Rong Jin, and Zhi-Hua Zhou. Speedup matrix completion with side information: Application to multi-label learning. In *Advances in Neural Information Processing Systems*, pages 2301–2309, 2013.

[16] S. Lakshminarayana. Categorization of web pages ˘2013 performance enhancement to search engine. *Knowledge-Based Systems*, 22(1):100 – 104, 2009.

[17] Carlos Castillo, Debora Donato, Aristides Gionis, Vanessa Murdock, and Fabrizio Silvestri. Know your neighbors: Web spam detection using the web topology. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 423–430. ACM, 2007.

[18] Björn H Junker and Falk Schreiber. *Analysis of biological networks*, volume 2. John Wiley & Sons, 2011.

[19] Dennis M Wilkinson and Bernardo A Huberman. A method for finding communities of related genes. *proceedings of the national Academy of sciences*, 101(suppl 1):5241–5248, 2004.

[20] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[21] Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331, 2008.

[22] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 225–236. IEEE, 2013.

[23] Dominique LaSalle, Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, George Karypis, and Pradeep Dubey. Improving graph partitioning for modern graphs and architectures. 2015.

[24] Dominique LaSalle and George Karypis. A parallel hill-climbing algorithm for graph partitioning. 2015. Technical Report 15-019.

[25] Dominique LaSalle and George Karypis. Efficient nested dissection for multicore architectures. In *Euro-Par 2015: Parallel Processing*, pages 467–478. Springer Berlin Heidelberg, 2015.

[26] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[27] Alan George and Joseph WH Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis*, 15(5):1053–1069, 1978.

[28] Dominique LaSalle and George Karypis. Multi-threaded modularity based graph clustering using the multilevel paradigm. *Journal of Parallel and Distributed Computing*, 2014.

[29] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[30] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153 – 159, 1992.

[31] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[32] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.

[33] MR Garey and DS Johnson. Computers and intractability: a guide to the theory of np-completeness. 1979.

[34] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning. *J. Global Optimization*, 29(2):225–241, 2004.

[35] Siew Yin Chan, Teck Chaw Ling, and Eric Aubanel. The impact of heterogeneous multi-core clusters on graph partitioning: an empirical study. *Cluster Computing*, 15(3):281–302, 2012.

[36] Vincent Heuveline. Hiflow 3: a flexible and hardware-aware parallel finite element package. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, page 4. ACM, 2010.

[37] Dénes König. *Theory of finite and infinite graphs*. Springer, 1990.

[38] Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 1998.

[39] Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[40] Reinhard Diestel. *Graph theory {graduate texts in mathematics; 173}*. Springer-Verlag Berlin and Heidelberg GmbH & amp, 2000.

[41] Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fundamenta mathematicae*, 1(15):271–283, 1930.

[42] Carlo Janna, Andrea Comerlati, and Giuseppe Gambolati. A comparison of projective and direct solvers for finite elements in elastostatics. *Advances in Engineering Software*, 40(8):675–685, 2009.

[43] Markus Wittmann and Thomas Zeiser. Technical note: Data structures of ilbdc lattice boltzmann solver. 2011.

[44] Adam Dziekonski, Adam Lamecki, and Michal Mrozowski. Tuning a hybrid gpu-cpu v-cycle multilevel preconditioner for solving large real and complex systems of fem equations. *Antennas and Wireless Propagation Letters, IEEE*, 10:619–622, 2011.

[45] THH Pian, Pin Tong, and CH Luk. Elastic crack analysis by a finite element hybrid method. Technical report, DTIC Document, 1971.

[46] Pin Tong, THH Pian, and S Jo Lasry. A hybrid-element approach to crack problems in plane elasticity. *International Journal for Numerical Methods in Engineering*, 7(3):297–308, 1973.

[47] Gary L. Miller, Shang hua Teng, William Thurston, and Stephen A. Vavasis. Geometric separators for finite-element meshes. *SIAM J. Sci. Comput*, 1998.

[48] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[49] Marián Boguñá, Romualdo Pastor-Satorras, Albert Díaz-Guilera, and Alex Arenas. Models of social networks based on social distance attachment. *Physical review E*, 70(5):056122, 2004.

[50] Mark EJ Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98(2):404–409, 2001.

[51] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[52] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the*

eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pages 177–187. ACM, 2005.

[53] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. 2012.

[54] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.

[55] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM, 2012.

[56] Joyce Jiyoung Whang, Xin Sui, and Inderjit S Dhillon. Scalable and memory-efficient clustering of large-scale social networks. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 705–714. IEEE, 2012.

[57] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[58] David F. Gleich. Hierarchical directed spectral graph partitioning. Information Networks, Stanford University, Final Project, 2005, 2006.

[59] Wikipedia. Most-referenced articles — Wikipedia, the free encyclopedia, 2015. [Online; accessed 10-October-2015].

[60] Oliver Marquardt and Stefan Schamberger. Open benchmarks for load balancing heuristics in parallel adaptive finite element computations. In *PDPTA*, pages 685–691, 2005.

[61] P Erdős and A Rényi. On random graphs i. *Publ. Math. Debrecen*, 6:290–297, 1959.

[62] B Delaunay. Sur la sphere vide. a la memoire de george voronoi, 1934.

[63] Peter Su and Robert L Scot Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 61–70. ACM, 1995.

[64] Paolo Cignoni, Claudio Montani, and Roberto Scopigno. Dewall: A fast divide and conquer delaunay triangulation algorithm in E-d. *Computer-Aided Design*, 30(5):333–341, 1998.

[65] Edward N Gilbert. Random plane networks. *Journal of the Society for Industrial & Applied Mathematics*, 9(4):533–543, 1961.

[66] Herbert A Simon. On a class of skew distribution functions. *Biometrika*, pages 425–440, 1955.

[67] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[68] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[69] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002.

[70] Jordi Duch and Alex Arenas. Community detection in complex networks using extremal optimization. *Physical review E*, 72(2):027104, 2005.

[71] A. van Heukelum, G. T. Barkema, and R. H. Bisseling. DNA Electrophoresis Studied with the Cage Model. *Journal of Computational Physics*, 180:313–326, 2002.

[72] Olaf Schenk, Andreas Wächter, and Martin Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2008.

[73] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, et al. Exascale computing study: Technology challenges in achieving exascale systems Peter Kogge, editor & study lead. 2008.

[74] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[75] Mark E Russinovich, David A Solomon, and Jim Allchin. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, volume 4. Microsoft Press Redmond, 2005.

[76] Richard McDougall and Jim Mauro. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture.* Pearson Education, 2006.

[77] Murray Stokely. *The FreeBSD Handbook 3rd Edition, Vol. 1: User Guide.* FreeBSD Mall, 2004.

[78] Michael Kerrisk. *The Linux programming interface.* No Starch Press, 2010.

[79] Johnson M Hart. *Windows system programming.* Pearson Education, 2010.

[80] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[81] Daniel Delling, Daniel Fleischman, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. An exact combinatorial algorithm for minimum graph bisection. *Mathematical Programming*, pages 1–42, 2014.

[82] William E Donath and Alan J Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, 1973.

[83] Alex Pothen, Horst D Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[84] Horst D Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135–148, 1991.

[85] Stephen T Barnard and Horst D Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and experience*, 6(2):101–117, 1994.

[86] Chengzhong Xu and Yibing Nie. Relaxed implementation of spectral methods for graph partitioning. In *Solving Irregularly Structured Problems in Parallel*, pages 366–375. Springer, 1998.

[87] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak mathematical journal*, 23(2):298–305, 1973.

[88] Daniela Calvetti, L Reichel, and Danny Chris Sorensen. An implicitly restarted lanczos method for large symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 2(1):21, 1994.

[89] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.

[90] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Labratories, 1993.

[91] Kirk Schloegel, George Karypis, Vipin Kumar, J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, and Morgan Kaufmann. Graph partitioning for high performance scientific simulations, 2000.

[92] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *ICPP (3)*, pages 113–122, 1995.

[93] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):69–79, 1999.

[94] Ilya Safro, Dorit Ron, and Achi Brandt. Multilevel algorithms for linear ordering problems. *J. Exp. Algorithmics*, 13:4:1.4–4:1.20, February 2009.

[95] Cédric Chevalier and Ilya Safro. Learning and intelligent optimization. chapter Comparison of Coarsening Schemes for Multilevel Graph Partitioning, pages 191–205. Springer-Verlag, Berlin, Heidelberg, 2009.

[96] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[97] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175 –181, june 1982.

[98] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 29. ACM, 1995.

[99] Silvio Micali and Vijay V Vazirani. An o (v— v— c— e—) algoithm for finding maximum matching in general graphs. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 17–27. IEEE, 1980.

[100] Harold N Gabow. *Data structures for weighted matching and nearest common ancestors with linking*. University of Colorado, Boulder, Department of Computer Science, 1990.

[101] David Avis. *Two greedy heuristics for the weighted matching problem*. MacGill University. School of Computer Science, 1977.

[102] Burkhard Monien, Robert Preis, and Ralf Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26(12):1609–1634, 2000.

[103] Jie Chen and Ilya Safro. Algebraic distance on graphs. *SIAM Journal on Scientific Computing*, 33(6):3468–3490, 2009.

[104] Jie Chen and Ilya Safro. A measure of the local connectivity between graph vertices. *Procedia Computer Science*, 4:196–205, 2011.

[105] Daniel A Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[106] Cédric Chevalier and Ilya Safro. Weighted aggregation for multi-level graph partitioning. *Combinatorial Scientific Computing*, (09061), 2008.

[107] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[108] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1055–1064, May 2015.

[109] Chris Walshaw and Mark Cross. Jostle: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques*, pages 27–58, 2007.

[110] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[111] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *ALENEX*, pages 16–29, 2012.

[112] Jianya Gong and Sung Kyu Lim. Multiway partitioning with pairwise movement. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 512–516. IEEE, 1998.

[113] Shantanu Dutt and Wenyong Deng. Vlsi circuit partitioning by cluster-removal using iterative improvement techniques. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 194–200. IEEE Computer Society, 1997.

[114] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magns Halldrsson, editors, *Algorithms - ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer Berlin / Heidelberg, 2011.

[115] Chris Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1-4):325–372, 2004.

[116] Charles J Alpert and Andrew B Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI journal*, 19(1):1–81, 1995.

[117] Duen-Ren Liu and Shashi Shekhar. Partitioning similarity graphs: a framework for declustering problems. *Information Systems*, 21(6):475–496, 1996.

[118] Ümit V Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 10(7):673–693, 1999.

[119] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[120] Aleksandar Trifunović and William J Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563–581, 2008.

[121] Sivasankaran Rajamanickam and Erik G Boman. Parallel partitioning with zoltan: Is hypergraph partitioning worth it? *Graph Partitioning and Graph Clustering*, 588:37–52, 2012.

[122] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[123] Bruce Hendrickson and Edward Rothberg. Effective sparse matrix ordering: Just around the bend. In *Proc. of 8th SIAM Conf. Parallel Processing for Scientific Computing*. Citeseer, 1997.

[124] William W. Hager and James T. Hungerford. Continuous quadratic programming formulations of optimization problems on graphs. *European Journal of Operational Research*, 240(2):328 – 337, 2015.

[125] William W Hager, James T Hungerford, and Ilya Safro. A multilevel bilinear programming algorithm for the vertex separator problem. *arXiv preprint arXiv:1410.4885*, 2014.

[126] Harry M Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[127] Joseph WH Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software (TOMS)*, 11(2):141–153, 1985.

[128] Pinar Heggernes, SC Eisestat, Gary Kumfert, and Alex Pothen. The computational complexity of the minimum degree algorithm. Technical report, DTIC Document, 2001.

[129] Alan George, Michael T Heath, Joseph Liu, and Esmond Ng. Sparse cholesky factorization on a local-memory multiprocessor. *SIAM journal on Scientific and Statistical Computing*, 9(2):327–340, 1988.

[130] Alan George, Joseph WH Liu, and Esmond Ng. Communication results for parallel sparse cholesky factorization on a hypercube. *Parallel Computing*, 10(3):287–298, 1989.

[131] José Ignacio Aliaga, Matthias Bollhöfer, Alberto F Martín, and Enrique S Quintana-Ortí. Evaluation of parallel sparse matrix partitioning software for parallel multilevel ilu preconditioning on shared-memory multiprocessors. In *PARCO*, pages 125–132, 2009.

[132] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[133] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[134] Ken Wakita and Toshiyuki Tsurumi. Finding community structure in a megascale social networking service. In *Proceedings of IADIS international conference on WWW/Internet 2007*, pages 153–162, 2007.

[135] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[136] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[137] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. Community detection in social media. *Data Mining and Knowledge Discovery*, 24(3):515–554, 2012.

[138] Jason Riedy, David A Bader, and Henning Meyerhenke. Scalable multi-threaded community detection in social networks. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1619–1628. IEEE, 2012.

[139] Bas Fagginger Auer and Rob H Bisseling. Graph coarsening and clustering on the gpu. *Graph Partitioning and Graph Clustering*, 588:223, 2012.

[140] Christian L. Staudt and Henning Meyerhenke. Engineering high-performance community detection heuristics for massive graphs. In *proceedings of the 2013 International Conference on Parallel Processing*. Conference Publishing Services (CPS), 2013.

[141] Andreas Noack and Randolf Rotta. Multi-level algorithms for modularity clustering. In *Experimental Algorithms*, pages 257–268. Springer, 2009.

[142] Hristo N Djidjev and Melih Onus. Scalable and accurate graph clustering and community structure detection. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[143] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[144] Ümit V Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Ucar. Multithreaded clustering for multi-level hypergraph partitioning. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 848–859. IEEE, 2012.

[145] Intel. *Intel OpenMP Runtine Library*, 2014.

[146] François Pellegrini. PT-Scotch and libScotch 5.1 User's Guide, August 2008. 76 pages User's manual.

[147] I. S. Duff and J. K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 22(2):227–257, June 1996.

[148] Bruce Hendrickson and Edward Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, 20(2):468–489, 1998.

[149] Free Software Foundation. *The GNU OpenMP Implementation*, 2014.

[150] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 20(2):172–188, 2008.

[151] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Priority queues. The MIT Press, 3rd edition, 2009.

[152] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[153] Michael Ovelgönne and Andreas Geyer-Schulz. An ensemble learning strategy for graph clustering. In *Graph Partitioning and Graph Clustering*, pages 187–206. American Mathematical Society, 2013.