Parallel and External High Quality Graph Partitioning

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Yaroslav Akhremtsev

aus Rostow am Don

Tag der mündlichen Prüfung: 29.05.2019

Erster Gutachter: Herr Prof. Dr. Peter Sanders

Zweiter Gutachter: Herr Prof. Dr. Henning Meyerhenke

Dedication to my mother, my grandmother, and my father.

Abstract

Partitioning graphs into k blocks of roughly equal size such that few edges run between the blocks is a key tool for processing and analyzing large complex real-world networks. The graph partitioning problem has multiple practical applications in parallel and distributed computations, data storage, image processing, VLSI physical design and many more. Furthermore, recently, size, variety, and structural complexity of real-world networks has grown dramatically. Therefore, there is a demand for efficient graph partitioning algorithms that fully utilize computational power and memory capacity of modern machines.

A popular and successful heuristic to compute a high-quality partitions of large networks in reasonable time is *multi-level graph partitioning* approach which contracts the graph preserving its structure and then partitions it using a complex graph partitioning algorithm. Specifically, the multi-level graph partitioning approach consists of three main phases: coarsening, initial partitioning, and uncoarsening. During the coarsening phase, the graph is recursively contracted preserving its structure and properties until it is small enough to compute its initial partition during the initial partitioning phase. Afterwards, during the uncoarsening phase the partition of the contracted graph is projected onto the original graph and refined using, for example, local search.

Most of the research on heuristical graph partitioning focuses on sequential algorithms or parallel algorithms in the distributed memory model. Unfortunately, previous approaches to graph partitioning are not able to process large networks and rarely take in into account several aspects of modern computational machines. Specifically, the amount of cores per chip grows each year as well as the price of RAM reduces slower than the real-world graphs grow. Since HDDs and SSDs are $50 - 400$ times cheaper than RAM, external memory makes it possible to process large real-world graphs for a reasonable price. Therefore, in order to better utilize contemporary computational machines, we develop efficient *multi-level graph partitioning* algorithms for the shared-memory and the external memory models.

First, we present an approach to shared-memory parallel multi-level graph partitioning that guarantees balanced solutions, shows high speed-ups for a variety of large graphs and yields very good quality independently of the number of cores used. Important ingredients include parallel label propagation for both coarsening and uncoarsening, parallel initial partitioning, a simple yet effective approach to parallel localized local

search, and fast locality preserving hash tables that effectively utilizes caches. The main idea of the parallel localized local search is that each processors refines only a small area around a random vertex reducing interactions between processors. For example, on 79 cores, our algorithms partitions a graph with more than 3 billions of edges into 16 blocks cutting 4.5% less edges than the closest competitor and being more than two times faster. Furthermore, another competitors is not able to partition this graph.

We then present an approach to external memory graph partitioning that is able to partition large graphs that do not fit into RAM. Specifically, we consider the semi-external and the external memory model. In both models a data structure of size proportional to the number of edges does not fit into the RAM. The difference is that the former model assumes that a data structure of size proportional to the number of vertices fits into the RAM whereas the latter assumes the opposite. We address the graph partitioning problem in both models by adapting the size-constrained label propagation technique for the semi-external model and by developing a size-constrained clustering algorithm based on graph coloring in the external memory. Our semi-external size-constrained label propagation algorithm (or external memory clustering algorithm) can be used to compute graph clusterings and is a prerequisite for the (semi-)external graph partitioning algorithm. The algorithms are then used for both the coarsening and the uncoarsening phase of a multi-level algorithm to compute graph partitions. Our (semi-)external algorithm is able to partition and cluster huge complex networks with billions of edges on cheap commodity machines. Experiments demonstrate that the semi-external graph partitioning algorithm is scalable and can compute high quality partitions in time that is comparable to the running time of an efficient internal memory implementation. A parallelization of the algorithm in the semi-external model further reduces running times.

Additionally, we develop a speed-up technique for the hypergraph partitioning algorithms. Hypergraphs are an extension of graphs that allow a single edge to connect more than two vertices. Therefore, they describe models and processes more accurately additionally allowing more possibilities for improvement. Most multi-level hypergraph partitioning algorithms perform some computations on vertices and their set of neighbors. Since these computations can be super-linear, they have a significant impact on the overall running time on large hypergraphs. Therefore, to further reduce the size of hyperedges, we develop a pin-sparsifier based on the min-hash technique that clusters vertices with similar neighborhood. Further, vertices that belong to the same cluster are substituted by one vertex, which is connected to their neighbors, therefore, reducing the size of the hypergraph. Our algorithm sparsifies a hypergraph such that the resulting graph can be partitioned significantly faster without loss in quality (or with insignificant loss). On average, `KaHyPar` with sparsifier performs partitioning about 1.5 times faster while preserving solution quality if hyperedges are large.

All aforementioned frameworks are publicly available.

Deutsche Zusammenfassung

Die Partitionierung von Graphen in k Blöcke von etwa gleicher Größe, sodass nur wenige Kanten zwischen den Blöcken verlaufen, ist ein wichtiges Werkzeug zur Verarbeitung und Analyse großer komplexer realer Netzwerke. Das Problem der Graphpartitionierung findet mehrere praktische Anwendungen in parallelen und verteilten Rechensystemen, bei der Datenspeicherung, Bildverarbeitung, physischen Gestaltung von VLSI sowie in vielen weiteren Bereichen. Weiterhin sind die Größe, Vielfalt und strukturelle Komplexität von realen Netzwerken in letzter Zeit enorm gestiegen. Daher besteht ein Bedarf an effizienten Algorithmen zur Graphpartitionierung, die die Rechenleistung und Speicherkapazität moderner Rechnern voll ausschöpfen. Eine verbreitete und erfolgreiche heuristische Methode, um qualitativ hochwertige Partitionen großer Netzwerke in angemessener Zeit zu berechnen, ist der Ansatz der mehrstufigen Graphpartitionierung, der den Graphen unter Beibehaltung seiner Struktur zusammenfasst und ihn dann unter Verwendung eines komplexen Partitionsalgorithmus für Graphen partitioniert. Konkret besteht der Ansatz der mehrstufigen Graphpartitionierung aus drei Hauptphasen: Vergröberung, anfängliche Partitionierung und Verfeinerung. Während der Vergröberungsphase wird der Graph rekursiv zusammengefasst, wobei seine Struktur und Eigenschaften erhalten bleiben, bis er klein genug ist, um seine anfängliche Partition während der anfänglichen Partitionierungsphase zu berechnen. Anschließend wird während der Vergröberungsphase die Partition des zusammengefassten Graphen auf den ursprünglichen Graphen abgebildet und beispielsweise durch eine lokale Suche verfeinert. Der größte Teil der Forschung zur heuristischen Graphpartitionierung legt den Schwerpunkt auf sequentielle oder parallele Algorithmen im verteilten Speichermodell. Leider sind bisherige Ansätze zur Graphpartitionierung nicht auf dem Stand, um große Netzwerke zu verarbeiten und berücksichtigen selten mehrere Aspekte moderner Rechenmaschinen. Insbesondere wächst die Anzahl der Transistoren pro Chip jedes Jahr, währenddessen der RAM-Preis langsamer sinkt als der reale Graphen-Wachstum. Da HDD-und SSD-Speicher 50 bis 400 mal billiger sind als RAM, ermöglicht der Einsatz des externen Speichers die Verarbeitung großer realer Grafiken zu einem vernünftigen Preis. Deshalb entwickeln wir effiziente Algorithmen der mehrstufigen Graphpartitionierung für das Shared Memory- und die externen Speichermodelle, um die Kapazitäten moderner Rechner optimaler nutzen zu können. Zuerst stellen wir einen Ansatz für die parallele mehrstufige Graphpartitionierung im Shared Memory vor, der ausgewogene Lösungen garantiert, signifikante Beschleunigungen für eine Vielzahl von großen Graphen bietet und unabhängig von der

Anzahl der verwendeten Transistoren eine sehr gute Qualität gewährleistet. Wichtige Bestandteile dieses Ansatzes umfassen die parallele Label Propagation sowohl für die Vergröberung als auch für die Verfeinerung sowie die parallele initiale Partitionierung. Diese ist ein einfacher, aber wirkungsvoller Ansatz zur parallelen lokalisierten lokalen Suche und zu schnellen Hash-Tabellen, die Lokalitäten erhalten und Caches effektiv nutzen. Die Grundidee der parallelen lokalisierten lokalen Suche besteht darin, dass jeder Prozessor nur einen kleinen Bereich um einen zufälligen Knoten herum verfeinert, wodurch die Interaktionen zwischen den Prozessoren reduziert werden. So partitionieren unsere Algorithmen beispielsweise auf 79 Transistoren einen Graph mit mehr als 3 Milliarden Kanten in 16 Blöcke, die 4.5% weniger Kanten schneiden als der stärkste Wettbewerber. Dabei erfolgen die Berechnungen mehr als doppelt so schnell. Darüber hinaus sind andere Wettbewerber nicht in der Lage, diesen Graphen zu partitionieren. Anschließend stellen wir einen Ansatz für die Partitionierung von Graphen des externen Speichers vor, der in der Lage ist, große Graphen zu partitionieren, die nicht in den RAM passen. Insbesondere gehen wir auf das externe und das semi-externe Speichermodell ein. Bei den beiden Modellen passt eine Datenstruktur einer Größe proportional zur Anzahl der Kanten nicht in den RAM. Der Unterschied liegt darin, dass das erstere Modell davon ausgeht, dass eine Datenstruktur mit einer Größe proportional zur Anzahl der Knoten in den RAM passt, während das letztere das Gegenteil annimmt. Wir befassen uns mit dem Problem der Graphpartitionierung in beiden Modellen, indem wir die Technik der größenbeschränkten Label Propagation für das semi-externe Modell anpassen und einen Algorithmus für das größenbeschränkte Clustering entwickeln, der auf der Graphfärbung im externen Speicher basiert. Unser semi-externer, größenbeschränkter Label-Propagierung-Algorithmus (oder externer Speicher-Clustering-Algorithmus) kann zur Berechnung von Graph-Clusterings verwendet werden und ist eine Voraussetzung für den (semi-)externen Graph-Partitionierung-Algorithmus. Die Algorithmen werden dann sowohl für die Vergröberungs- als auch für die Verfeinerungsphase eines mehrstufigen Algorithmus zur Berechnung von Graphpartitionen verwendet. Unser (semi-)externer Algorithmus ist in der Lage, riesige komplexe Netzwerke mit Milliarden von Kanten auf günstigen Standardrechnern zu partitionieren und in Cluster zu verpacken. Versuche zeigen, dass der semi-externe Graphpartitionierung-Algorithmus skalierbar und in der Lage ist, qualitativ hochwertige Partitionen innerhalb von Zeitspannen zu berechnen, die mit der Laufzeit einer effizienten internen Speicherimplementierung vergleichbar sind. Durch eine Parallelisierung des Algorithmus im semi-externen Modell werden die Laufzeiten zusätzlich reduziert. Zusätzlich entwickeln wir eine beschleunigte Technik für die Hypergraph-Partitionierung-Algorithmen. Hypergraphen sind eine Erweiterung von Graphen, die es einer einzelnen Kante ermöglichen, mehr als zwei Knoten zu verbinden. Sie beschreiben daher Modelle und Prozesse genauer und bieten darüber hinaus mehr Optimierungsmöglichkeiten. Die meisten mehrstufigen Algorithmen zur Hypergraph-Partitionierung führen bestimmte Berechnungen an Knoten und an den benachbarten Gruppen durch. Da diese Berechnungen superlinear sein können, haben sie einen signifikanten Einfluss auf die gesamte Laufzeit großer Hypergraphen. Um die Größe der Hyper-Kanten weiter zu reduzieren, entwickeln wir eine Pin-Verdichtung, der

auf dem MinHash-Verfahren basiert, das Knoten bündelt, die ähnliche Nachbarn haben. Weiterhin werden Knoten, die zum gleichen Cluster gehören, durch einen Knoten ersetzt, der mit ihren Nachbarn verbunden ist, wodurch die Größe des Hypergraphen reduziert wird. Unser Algorithmus verdichtet einen Hypergraphen so, dass der entstandene Graph ohne Qualitätsverlust (oder mit einem geringfügigem Qualitätsverlust) wesentlich schneller partitioniert werden kann. Im Durchschnitt führt KaHyPar mit Verdichtung die Partitionierung in etwa 1.5 mal schneller durch, wobei die Qualität der Lösung erhalten bleibt. Alle oben genannten Frameworks sind öffentlich zugänglich.

Acknowledgements

Here I would like to thank all people who supported me during writing of this thesis. First of all, I'm very thankful to my supervisor Peter Sanders. I thank him a lot for all opportunities he gave me to perform research in computer science. I would like to thank my former and current colleagues for all interesting discussions we had and their help in research. Thank you Michael Axtmann, Tomas Balyo, Timo Bingmann, Daniel Funke, Demian Hespe, Lorenz Hübschle-Schneider, Sebastian Lamm, Tobias Maier, Vitaly Osipov, Sebastian Schlag, Dominik Schreiber, Darren Strash, Sascha Witt. Expect a lot of coffee and cakes as a symbol of gratitude. I also would like to thank my mother, grandmother, and father for all love and support they give me. Without it I doubt this thesis would be ever finished. Finally, last but not least, I thank my best friends Grigoriy Kolosov and Anastasia Dudkina for their support and love. And although we live far away from each other, our friendship grows closer and closer from year to year.

Table of Contents

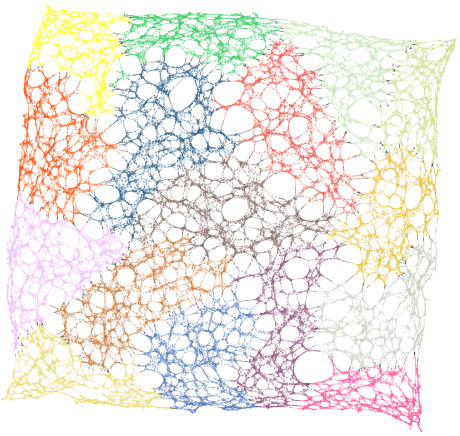
Chapter 1

1 Introduction

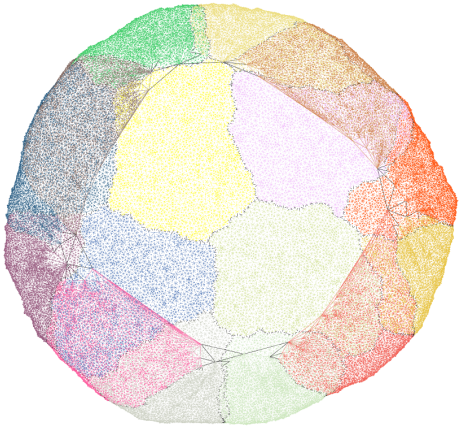
Graph partitioning is a fundamental combinatorial problem that has multiple practical applications in parallel and distributed computations, data storage, image processing, VLSI physical design and many more [Bul+13]. Specifically, when one needs a balanced distribution of data or computations over machines minimizing communication between them to save running time, one may consider to solve a graph partitioning problem. The first example is an efficient distribution of data over machines in a data center that minimizes the average number of machines accessed per query. For example, if the social graph (users are vertices and friendship defines edges between them) is stored in a data center that consists of multiple machines, we want to store the user data preserving its locality. Meaning that when answering a user query, we want to minimize the number of machines that are accessed since we often need to access the user data of friends. Specifically, we want as many friends to be on the same machine as possible given that the distribution of the users between machines must be balanced. Therefore, if the data center has k machines, we need to partition the social graph into k parts of about equal size. Furthermore, we want to minimize the sum of the edges that run between the machines. Another example is distributed iterative numerical methods that solve partial differential equations. These equations are discretized over a mesh where each node represents a portion of computational work and edges define information flow between nodes. To map a mesh onto k machines, we need to solve a graph partitioning problem since we want to simultaneously balance computational work between machines and minimize the amount of communication after each iteration. Figures 1.1 (a)– 1.1 (c) show different partitioned graphs.

Unfortunately, the graph partitioning problem is NP-hard [GJS74; HR73]. Meaning that there are no known polynomial time solutions for this problem. Therefore, heuristical algorithms are necessary to partition large real-world networks in reasonable time. There is a large variety of different approaches to solve this problem. One of the most promising and popular heuristics, which yields a good trade-off between running time and quality in practice, is *multi-level graph partitioning*. This can be explained by the fact that *multi-level graph partitioning* combines both global and local views on the graph allowing more flexibility in searching for a good solution. It consists of three main phases: *coarsening*, *initial partitioning* and *uncoarsening*. The main idea is to

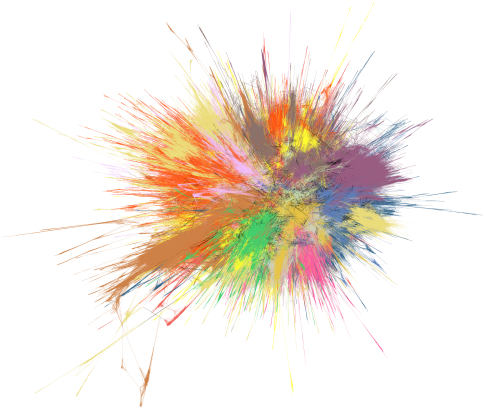
recursively contract (shrink) the graph preserving its original structure, constructing a hierarchy of contracted graphs (*coarsening*). When the coarsest graph is small enough, a more sophisticated but slower method computes an *initial partition*. Afterwards, the coarsest graph is uncontracted, its partition is projected onto the predecessor graph in the hierarchy and improved using refinement techniques (*uncoarsening*).



(a) Random geometric graph.



(b) Delaunay graph.



(c) Graph of Amazon in 2008.

Figure 1.1: Partitionings of different graphs into 16 blocks. Each color correspond to a block.

Most of the research on heuristical graph partitioning focuses on sequential algorithms or parallel algorithms in the distributed memory model. Both models do not take into account several aspects of modern computational machines. Specifically, the amount

of cores per chip grows each year and the external memory (e.g. HDDs and SSDs) reduces. Therefore, there is a need to efficiently utilize multiple cores and external memory in computations. In order to do this, we develop efficient multi-level graph partitioning algorithms for the shared-memory and the external memory models that are able to partition large real-world graphs.

The first computational model we consider is the shared-memory model. Although Moore's law [Moo98] declares that the number of transistors approximately doubles every two years, the average clock frequency of processors has stagnated in the last ten years, whilst the number of cores per chip has been drastically increasing. Thus, there is demand for parallel shared-memory graph partitioning algorithms that efficiently utilize available computing power. Although there are several shared-memory graph partitioning frameworks, many possibilities for further improvement remain, especially when one needs a fast *high-quality* graph partitioner. Furthermore, already existing parallel distributed memory graph partitioning algorithms do not take into account several important features of the shared-memory model like availability of caches or shared memory. An efficient utilization of these features allows to improve performance and simplify graph partitioning algorithms.

The second computational model we consider is the external memory model. This model is used to develop and analyze efficient algorithms that store most data on hard disk drives (HDD) or solid-state drives (SSD). Since real-worlds graphs grow faster than the available size of RAM, using efficient graph partitioning algorithms which use HDDs or SSDs is the only way to partition large graphs on cheap commodity servers and machines. For example, the Facebook social graph [WA] increased in size from 10 GB to 1 TB between 2007 and 2011. Thus, its size increases by a factor of 2.5 per year, whereas the price of RAM on average decreases by a factor of 1.3 per year, and HDDs and SSDs are always $50 - 400$ times cheaper than RAM. As a result, external memory is very cheap and makes it possible to process large amounts of data which is impossible using only RAM of cheap commodity machines and servers or too expensive on large servers. Furthermore, there is *no* previous work on external memory graph partitioning algorithms to the best of our knowledge.

Additionally, we consider the hypergraph partitioning problem. Hypergraphs are a generalization of graphs that allow a single edge to connect *more than two* vertices. Thus, they describe models and processes more accurately additionally allowing more possibilities for improvement. For example, hypergraphs can be used to model integrated circuits or sparse matrices. Specifically, partitioning hypergraphs that model integrated circuits has variety of application in the VLSI domain (e.g., design packaging and optimization). Moreover, a partition of a hypergraph that models a sparse matrix allows to find a reordering of its rows and columns that speeds up algebraic computations on this matrix in a distributed setting. Most multi-level hypergraph partitioning algorithms perform computations on vertices and their corresponding set of neighbors. For large hypergraphs, these calculations might significantly affect the overall running time. Therefore, we optimize a high-quality multi-level algorithm that

partitions hypergraphs into k balanced blocks. Specifically, our algorithm sparsifies a hypergraph such that the resulting graph can be partitioned significantly faster without loss in quality (or with insignificant loss). Although the main focus of this thesis is graph partitioning, we believe that the presented algorithm and underlying techniques are of great interest and can be applied in graph partitioning as well.

1.1 Main Contributions

Our first contribution is a parallel shared-memory graph partitioning framework called `Mt-KaHIP` (based on `KaHIP` [SS11] – Karlsruhe High Quality Partitioning) that constructs *high-quality* partitions and shows *good scalability*. To achieve better running times, we parallelized and optimized all three phases of the multi-level graph partitioning approach. Furthermore, we introduced additional improvements to each of the phases that result in partitions of better quality. The main components of coarsening are *parallel label propagation* [RAK07] and *parallel contraction*. Specifically, we developed a novel load balancing technique for the parallel label propagation algorithm that allows to efficiently process graphs with arbitrary vertex degree distributions. To contract the graph in parallel, we use a concurrent hash table. In the initial partitioning phase, we partition the coarse graph *multiple times* in parallel to improve quality. Finally, we use a combination of *parallel label propagation* and *parallel localized local search* to improve partition quality. The latter approach allows to find *high-quality balanced* partitions while scaling better than competitors. The main idea of parallel localized local search is that each processor refines only a small area around a random vertex and, thus, processors interact rarely. Specifically, each processor changes the partition only locally and the partition is changed globally in a sequential way only after each processor finishes. This separation allows to efficiently parallelize the search for possible improvements of the partition which is usually the most time consuming part of parallel localized local search. An important ingredient of the parallel localized local search is fast locality preserving hash tables that efficiently utilize caches. These hash tables store integer keys such that keys with small absolute differences are within the same cache line. Therefore, if accesses to keys occur in a local manner we expect that the number of cache misses reduces. Furthermore, we perform additionally optimizations to avoid false sharing and decrease negative effects of non-uniform memory accesses (NUMA). As a result, our graph partitioning framework partitions large real-world graphs with *better quality and scalability* than other parallel graph partitioning frameworks. We test our framework and its competitors on the benchmark that includes graphs with up to 3.3G edges. For example, it partitions a graph of some 3.3G edges in about 42s with better quality than other graph partitioning frameworks. Furthermore, one of our main competitors cannot partition the graph whilst another one produces a partition with a 4.5% larger cut size spending about 100s.

Our second contribution is an external memory graph partitioner that partitions large graphs which do no fit into RAM. Specifically, we consider two types of problems: the semi-external problem and the external problem. In both problems a data structure of the size proportional to the number of edges *does not* fit into RAM. The difference is that the former problem assumes that a data structure of the size proportional to the number of vertices *fits* into the RAM whereas the latter assumes the *opposite*. We developed a semi-external size-constrained label propagation algorithm [RAK07] and a external label propagation algorithm that constructs size-*un*constrained clusters using external memory priority queues and the time forward processing technique [Zeh02]. To be able to construct size-constrained clusters in the external memory model, we developed an external memory clustering algorithm that uses graph coloring and time forward processing to maintain up-to-date sizes of clusters. The main component of our framework is the *semi-external* size-constrained label propagation algorithm (or the clustering algorithm based on graph coloring) that constructs a clustering of the graph according to which it is contracted. We contract the graph until it is small enough to fit into RAM and then use the graph partitioning framework `KaHIP` to partition it in RAM. We additionally refine the partition using *(semi-)external* size-constrained label propagation (or the clustering algorithm based on graph coloring). We test our semi-external graph partitioning framework on a benchmark set of graphs with up to 80.5G edges. For example, our graph partitioner is able to partition a graph with 80.5G edges, which occupies of some 1.2TB of memory, on a single machine with only 128GB of RAM in about two hours, whereas our competitors are not able to process this graph.

Our third contribution is a hypergraph sparsifier that preserves the hypergraph's original structure. In order to sparsify hypergraphs, we combine vertices that share a lot of hyperedges. To find such vertices in linear time with a guaranteed probability, we use the min-hash technique by Broder [Bro97] that approximates the Jaccard index of hyperedges incident to any pair of vertices. But note that the Jaccard index for two vertices depends on the input hypergraph. Moreover, even in one hypergraph the Jaccard index for two vertices may be heterogeneous. Specifically, combining pairs of vertices using the same threshold for the Jaccard index may produce a hypergraph that does not preserve the original structure or only a small number of vertices is contracted. Therefore, we developed an adaptive sparsifier that repeatedly applies the min-hash technique to find pairs of neighbors with the Jaccard index within a preset interval. We implemented our sparsifier within the hypergraph partitioning framework `KaHyPar` (Karlsruhe Hypergraph Partitioning) [Akh+17]. On average, `KaHyPar` with sparsifier performs partitioning about 1.5 times faster while preserving solution quality if hyperedges are large.

All aforementioned frameworks are publicly available and can be found [Myg].

1.2 Outline

Chapter 2 presents basic definitions and concepts that we use in this thesis. Next, Chapter 3 presents an overview of existing graph partitioning techniques and algorithms. We specifically focus on algorithms that are used in the parallel graph partitioning frameworks. Chapter 4 presents our parallel shared-memory graph partitioning framework and Chapter 5 presents our (semi)-external graph partitioning framework. Afterwards, we present our sparsifier for hypergraphs in Chapter 6. The discussion of the results is presented in the end of each corresponding chapter as well as in Chapter 7.

Chapter 2

Preliminaries

2

In this chapter, we present basic definitions and concepts that are used throughout this thesis.

2.1 Graph Related Definitions

Let $G = (V, E, c, \omega)$ is a undirected graph where $V = \{0, \ldots, n-1\}$ is a set of vertices, $E \subseteq \{(v, w) : v, w \in V\}$ is a set of edges, $c : V \to \mathbb{R}_{>0}$ is a vertex weight function, and $\omega : E \to \mathbb{R}_{>0}$ is an edge weight function. Here (v, u) denotes a set of two vertices and $(v, u) = (u, v)$ since we consider undirected graphs. We extend edge and vertex weight functions ω and c to sets, e.g., $\omega(M) := \sum_{e \in M} \omega(e)$. The set $N(v) := \{u \in V : (v, u) \in E\}$ denotes the *neighbors* of v (*adjacency list* of v). The *degree* of a vertex v is $d(v) := |N(v)|$. The *maximum vertex degree* is Δ. The *density* of a graph is $2|E|/(|V|(|V|-1))$. A *subgraph* $G' = (V', E')$ of a graph G is a graph such that $V' \subseteq V$ and $E' \subseteq E$. Furthermore, G' is an induced subgraph of G if $E' = \{(v, u) \in E : v, u \in V'\}$. A matching M is a subset of edges with disjoint vertices. More formally, $M \subseteq E$ such that $\forall\ e, e' \in M : e \cap e' = \emptyset$. A maximum weighted matching is a matching such that it has a maximum weight over all possible matchings; that is, $\arg\max_M \omega(M)$. A set of vertices I is called an independent if $\forall\ v, u \in I : (v, u) \notin E$. Furthermore, a partition of V into k disjoint independent sets $C_1, \ldots, C_k$ is called *coloring* of the graph G, where each independent set C_i corresponds to a color.

2.1.1 Graph Partitioning and Clustering

The *graph partitioning problem* asks to partition vertices of a graph $G = (V, E)$ into k disjoint *subsets* (*blocks*) while minimizing a given objective function such that the total weight of each block does not exceed the *maximum weight* of a block $L_{\max} := (1 + \epsilon)\lceil |V|/k \rceil$ (for unit weights of vertices) for some imbalance parameter $\epsilon \in (0, 1)$. For non-unit weights of vertices $L_{\max} := (1 + \epsilon)\lceil c(V)/k \rceil + \max_{v \in V} c(v)$.

More formally, one needs to construct k subsets $V_1, \ldots, V_k$ of V such that an objective function f is minimized, $V_1 \cup \cdots \cup V_k = V$, $V_i \cap V_j = \emptyset \; \forall i, j$ and $\forall\, V_i : c(V_i) \leq L_{\max}$. We call a block V_i *overloaded* if its weight exceeds $L_{\max}$. We denote the block of vertex v as $B[v]$. We refer to the number of a block as partition ID. A set of edges that run between two blocks is called *cut*. More formally, let $E_{ij} := \{(u, v) \in E : u \in V_i, v \in V_j\}$ then the cut is $\cup_{i<j} E_{ij}$. A vertex $v \in V_i$ is a *boundary vertex* if it is adjacent to a vertex in a different block; that is, $\exists\, u \in N(v) : u \in V_j, j \neq i$. The most common objective function is the size of the *cut*, that is, $\omega(\cup_{i<j} E_{ij})$. We define the *gain of a vertex* as the maximum reduction in the cut size after moving it to a different block. Specifically, the gain of $v \in V_i$ is

$$g(v) := \max_{t \neq i} \omega(\{(v, w) : w \in N(v) \cap V_t\}) - \omega(\{(v, w) : w \in N(v) \cap V_i\})).$$

A k *vertex separator* is a set of vertices whose removal leaves in the graph at least k connected components.

A *clustering* is also a partition of the vertices. In the context of clustering, we refer to blocks as clusters. However, k is usually not given in advance and the balance constraint is removed. A *size-constrained clustering* contains only clusters of size at most U. Note that by adjusting the upper bound U one can control the number of clusters in a feasible clustering to some extent.

An abstract view of a clustered (partitioned) graph is a *quotient graph*, in which vertices represent blocks and edges are induced by connectivity between blocks. The quotient graph is the result of the contraction of the graph according to the clustering. The *weighted* version of the quotient graph has vertex weights that are set to the weight of the corresponding block and edge weights that are equal to the weight of the edges that run between the respective blocks. More formally, given a graph clustering (or a graph partitioning) $V_1, \ldots, V_k$, the *quotient graph* defined as $\mathcal{Q} = (V_q, E_q, c_q, \omega_q)$, where $V_q = \{1, \ldots, k\}$ and $E_q = \{(i, j) \mid E_{ij} \neq \emptyset\}$, $\omega_q(i, j) = \sum_{e \in E_{ij}} \omega(e)$, $c_q(i) = \sum_{v \in V_i} c(v)$, $i, j \in V_q$. An example is shown in Figure 2.1.

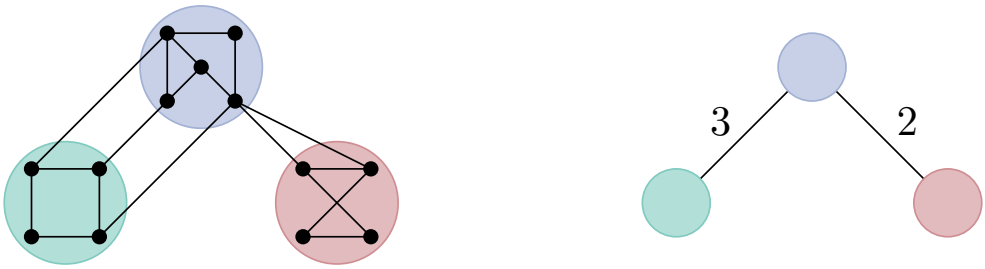


Figure 2.1: Clustering of a graph and the corresponding quotient graph. Here each color denotes a cluster of vertices in the original graph (left) and a vertex in the quotient graph (right).

All our input graphs G have unit edge weights and vertex weights. However, even those will be reduced to graph partitioning problems with weighted vertices during the course of the multi-level algorithm. In order to avoid a tedious notation, G will denote the current state of the graph before and after a (un)contraction in the multi-level scheme throughout our thesis.

2.2 Memory models

Here we describe memory models that we use in the analysis of our algorithms.

2.2.1 Random Access Machine and Parallel Random Access Machine

The *Random Access Machine* (RAM) [VN93] is a computational model that consists of a *processing element* (PE) and a random-access memory such that the PE can access a memory cell in constant time. The natural extension of RAM is a computational model with multiple PEs and a shared memory that can be accessed simultaneously by all PEs. We denote the *number of PEs* as p. If all PEs perform operations synchronously according to a common clock this computational model is called *Parallel Random Access Machine* (PRAM) [Wyl79]. In contrast, in the *asynchronous Parallel Random Access Machine* [GMR98] all PEs work asynchronously and it is responsibility of a programmer to ensure synchronization. Since several PEs can access the same memory cell simultaneously, there are three access patterns:

(i) *Exclusive Read Exclusive Write* (EREW) does not allow simultaneous accesses to the same memory cell.

(ii) *Concurrent Read Exclusive Write* (CREW) allows simultaneous read operations to the same memory cell but prohibits simultaneous write operations.

(iii) *Concurrent Read Concurrent Write* (CRCW) allows simultaneous read and write operations to the same memory cell. There are three strategies to resolve write conflicts. The *common* strategies allows simultaneous writes only if all of them attempt to write the same value. The *arbitrary* strategy allows an arbitrary write operation to succeed. The *priority* strategy assigns each PE a priority such that among multiple PEs, which attempt to write to the same memory, the PE with highest priority succeeds whereas others fail.

The complexity of concurrent write in the CRCW PRAM model if p PEs write to the same memory cell is $\Theta(1)$. Thus, this model does not take into account contention. However, when p PEs write to the same memory cell on a modern machine this will take $\Omega(p)$ time. Therefore, we consider *asynchronous Concurrent Read Queue Write*

(aCRQW) model [GMR98] that takes contention into account. Specifically, p PEs write to the same memory cell in $\Omega(p)$ time in the aCRQW model.

In this thesis, we use the *aCRQW* model to analyze algorithms and present for each *work* and *parallel time* complexities. Here work is the number of operations (as function of the input size) needed by a single PE to execute an algorithm and parallel time is the number of operations needed by p PEs to execute the same algorithm. We assume a *CRCW* access pattern and resolve conflicts using the *arbitrary* strategy. Namely, there is no any guarantee which value will be written. Furthermore, we allow to perform an *atomic concurrent update* of a memory cell using *compare and swap operation* `CAS`(x, y, z). If $x = y$ then this operation assigns $x \leftarrow z$ and returns `True`; otherwise it returns `False`.

Graph Data Structure. We represent a graph $G = (V, E)$ in memory using three arrays `offset`, `edge`, and `vertex_weight`. This representation is called adjacency array. The array `edge` of size $|E|$ contains all adjacency lists of vertices and the array `offset` of size $|V| + 1$ contains offsets to an adjacency lists of all vertices in the `edge` array. Specifically, `edge`$[$`offset`$[v], \ldots,$ `offset`$[v+1]-1]$ is an adjacency list of a vertex v, where `edge`$[$`offset`$[v]+i]$ is a pair (u, w) where u is an i-th neighbor of v and $w = \omega(v, u)$. The array `vertex_weight` of size $|V|$ contains weights of all vertices, where the weight of a vertex v is `vertex_weight`$[v]$. Figure 2.2 outlines described data structure.

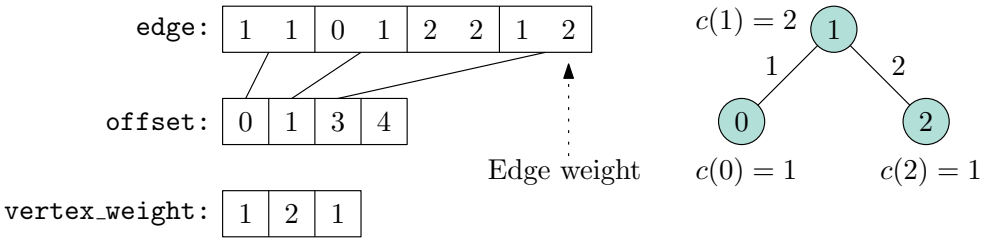


Figure 2.2: Graph represented using adjacency array.

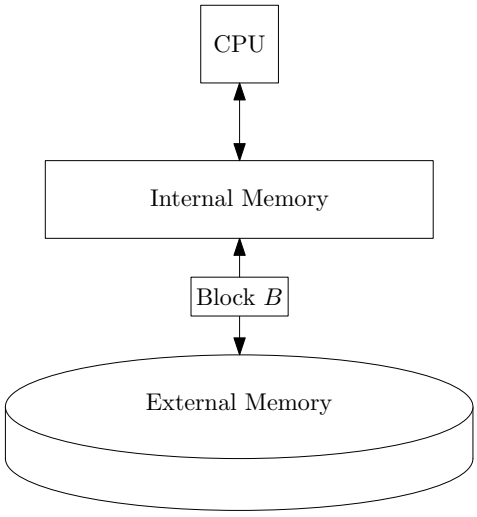
2.2.2 External Memory Model

We consider the *external memory model* [ABW02; Vit01; VS94] and solve two types of problems in this model: semi-external and external. In this model, theres is a limited amount of the internal memory M and an unlimited amount of the external memory. Input/output operations (I/O operations) read/write consecutive data in blocks of size B from/to the external memory to/from the internal memory. In practice, the external memory is kept on HDDs or SSDs and usually two cases are considered: the external memory model with one HDD or SSD (see Figure 2.3 (a)) and the external memory model with multiple HDDs or SSDs (see Figure 2.3 (b)). In the external memory

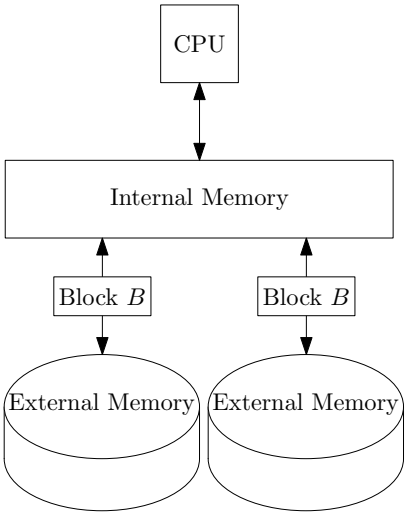
model an I/O operation is much more expensive than an access to the internal memory. Therefore, the goal is to minimize the total number of I/O operations. Basically, accesses to the internal memory are considered as free.

We will use the following notations for the external memory model with one HDD or SSD. The size of the internal memory is M, B is the size of a disk block, $\text{Scan}(N) = \mathcal{O}(N/B)$ is the number of I/O operations needed for reading or writing an array of size N and $\text{Sort}(N) = \mathcal{O}(N/B \log_{M/B} N/B)$ is the number of I/O operations needed for sorting an array of size N. For the external memory model with D HDDs or SSDs the I/O complexities are the following: $\text{Scan}(N) = \mathcal{O}(N/(BD))$ is the number of I/O operations needed for reading or writing an array of size N and $\text{Sort}(N) = \mathcal{O}(N/(BD) \log_{M/B} N/B)$ is the number of I/O operations needed for sorting an array of size N.

For both semi-external and external memory problems, we assume that $|E| = \Omega(|M|)$; that is, a data structure of size $\Omega(|E|)$ does not fit into the internal memory. The difference between two problems is in the size of the internal memory. In semi-external problems, we assume that $M = \Omega(|V|)$; i.e., any data structure of size $\mathcal{O}(|V|)$ fits into the internal memory, whereas in external problems, we assume that $|V| = \Omega(M)$. Namely, a data structure of size $\Omega(|V|)$ fits *only* in the external memory.



(a) An external memory model with one disk.



(b) An external memory model with two disks.

Prefetching. Prefetching [Dem06; Smi82] is a technique that allows to overlap input operations from disks with computations. More specifically, given a sequential access pattern to an array, prefetching allocates memory for at least two blocks in the internal memory and reads the next block into the internal memory while current block is

processed. If the time t_r to read a block is approximately equal to the time t_p to process a block then the time to scan an array of size N is $\text{Scan}(N) \cdot t_r$ whereas without prefetching it is $\text{Scan}(N) \cdot (t_r + t_p)$.

Graph Data Structure. We represent a graph in the external memory model using the adjacency array representation. To store neighbors of all vertices, we use an array `edge` of size $|E| + |V|$. This approach to store neighbors minimizes the number of I/O operations necessary to iterate over all edges ($\text{Scan}(|E|)$ I/O operations). An external array of the edges contains the adjacency lists of each vertex in increasing order of their IDs. Each element of the adjacency list of a vertex u is a pair (v, w), where v is the target of the edge (u, v) and $w = \omega(u, v)$ is the weight of the edge. We mark the end of each adjacency list by using the sentinel pair \$. This allows us to determine easily if we reached the end of the adjacency list of the vertex that we currently process. The second external array `offset` stores offsets of vertices, that is, for each vertex we store a pointer to the beginning of its adjacency list in the array `edges`. The third external array `vertex_weight` contains the weights of the vertices. Figure 2.4 outlines described data structure.

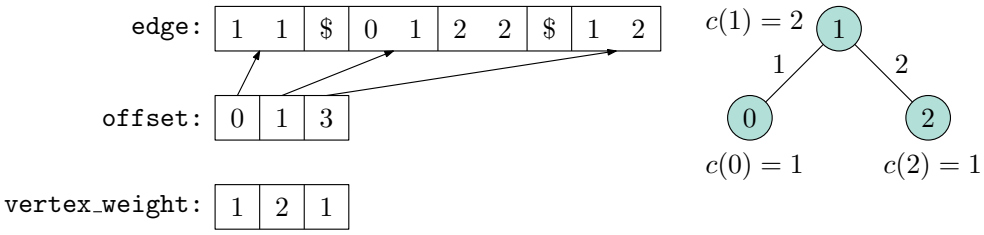


Figure 2.4: Graph represented using adjacency array in the external memory model.

2.3 Plots and Experimental Setup

In this section, we describe performance plots, the benchmark set of graphs and the computing machines we used in our experiments.

2.3.1 Performance plots

We use performance plots to present the quality comparison. A curve in a performance plot for algorithm X is obtained as follows. For each instance (graph and k), we calculate a normalized value $1 - \frac{\texttt{best}}{\texttt{cut}}$, where `best` is the best cut obtained by any of the considered algorithms and `cut` is the cut of algorithm X. These values are then

sorted. Thus, the result of the best algorithm is at the bottom of the plot. We set the value for the instance above 1 if an algorithm computes an imbalanced partition. Hence, it is in the top of the plot. Additionally, note that points that are on the same vertical line can correspond to different instances. Figure 2.5 shows an example of a performance plot.

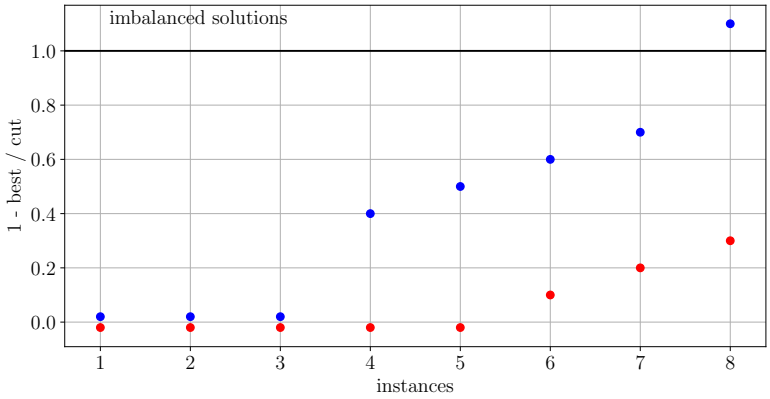


Figure 2.5: An example of a performance plot. We can see that Algorithm 1 computed the best cuts for 3 instances and one imbalanced partition. Algorithm 2 computed the best cuts for 5 instances.

2.3.2 Graph Families

We evaluate our algorithms on a benchmark set of large graphs. These graphs are collected from [Bad+13a; Bad+14; BV04; Dav; Les; LMP15]. Table 2.1 summarizes the main properties of the benchmark set. Our benchmark set includes a number of graphs from numeric simulations as well as complex networks (for the latter with a focus on social networks and web graphs).

We use five types of random graphs: `rgg`, `del`, `ba`, `er`, and `rhg` for comparisons. A graph `rgg`X is a *2D random geometric graph* with 2^X vertices where vertices represent random points in the (Euclidean) unit square and edges connect vertices whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost certainly connected. A graph `rgg`X`_3d` is a *3D random geometric graph* with 2^X vertices, where vertices represent random points in the (Euclidean) unit square and edges connect vertices whose Euclidean distance is below $0.55(\ln n/n)^{1/3}$. A graph `del`X is a 2D Delaunay triangulation of 2^X random points in the unit square. A graph `del`X`_3d` is a 3D Delaunay triangulation of 2^X random points in the unit square. We generate `rgg`X, `rgg`X`_3d`, `del`X, and

`delX_3d` graphs using KaGen [Fun+18]. The graph `ba_2_22` is a random graph generated using the Barabassi-Albert graph model $BA(n, d)$, where $n = 2^{22}$ and the minimum degree of a vertex (d) equals 2. The graph `er_2_22_2_23` is a random graph generated using Erdös-Rényi $G(n, m)$ model, where $n = 2^{22}$ and $m = 2^{23}$. The graph `er-fact1.5-scale23` is generated using the Erdös-Rényi $G(n, p)$ model with $p = 1.5 \ln n/n$. This threshold was chosen in order to ensure that the graph is almost certainly connected. The graph `rhg` is a complex network generated with NetworKit [LMP15] according to the *random hyperbolic graph* model [Kri+10]. In this model vertices are represented as points in the hyperbolic plane; vertices are connected by an edge if their hyperbolic distance is below a threshold. Additionally, we generated the graph `rhg_2_23` that is a random hyperbolic graph with the average degree of 8 using KaGen [Fun+18].

Table 2.1: Basic properties of the benchmark set with a rough type classification. C stands for complex networks, M is used for mesh type networks.

Graph	n	m	Type	Reference
amazon	≈0.4M	≈2.3M	C	[Les]
youtube	≈1.1M	≈3.0M	C	[Les]
amazon-2008	≈0.7M	≈3.5M	C	[WA]
ba_2_22	≈4.2M (2^{22})	≈8.4M	C	[Fun+18]
in-2004	≈1.4M	≈13.6M	C	[WA]
eu-2005	≈0.9M	≈16.1M	C	[WA]
er_2_22_2_23	≈4.2M (2^{22})	≈16.3 (2^{23})	M	[Fun+18]
packing	≈2.1M	≈17.5M	M	[Bad+14]
hugebubbles-00	≈18.3M	≈27.5M	M	[Bad+14]
rhg_2_23	≈8.4M (2^{23})	≈32.1M	M	[Fun+18]
com-LiveJournal	≈4M	≈34.7M	C	[LK14]
channel	≈4.8M	≈42.7M	M	[Bad+14]
cage15	≈5.2M	≈47.0M	M	[Bad+14]
ljournal-2008	≈5.4M	≈49.5M	C	[WA]
europe.osm	≈50.9M	≈54.1M	C	[Bad+13a]
enwiki-2013	≈4.2M	≈91.9M	C	[WA]
er-fact1.5-scale23	≈8.4M	≈100.3M	M	[Bad+13a]
hollywood-2011	≈2.2M	≈114.5M	C	[WA]
com-Orkut	≈3.1M	≈117.2M	C	[LK14]
enwiki-2018	≈5.6M	≈117.2M	C	[WA]
indochina-2004	≈7.4M	≈151.0M	C	[WA]
rhg	≈10.0M	≈199.6M	M	[LMP15]
del_2_26	≈67.1M (2^{26})	≈201.3M	M	[HSS10]
uk-2002	≈18.5M	≈261.8M	C	[WA]
del_2_27	≈134.2M (2^{27})	≈303.2M	M	[Fun+18]
nlpkkt240	≈28.0M	≈373.2M	M	[Dav]

rgg_2_26_3d	≈67.1M (2^{26})	≈379.6M	M	[Fun+18]
del_2_26_3d	≈67.1M	≈521.3M	M	[Fun+18]
arabic-2005	≈22.7M	≈553.9M	C	[WA]
rgg_2_26	≈67.1M (2^{26})	≈574.6M	M	[Fun+18]
uk-2005	≈39.5M	≈783.0M	C	[WA]
rgg_2_27_3d	≈134.2M (2^{27})	≈787.7M	M	[Fun+18]
webbase-2001	≈118.1M	≈854.8M	C	[WA]
it-2004	≈41.3M	≈1.0G	C	[WA]
del_2_27_3d	≈134.2M (2^{27})	≈1.0G	M	[Fun+18]
twitter-2010	≈41.7M	≈1.2G	C	[WA]
rgg_2_27	≈134.2M (2^{27})	≈1.2G	M	[Fun+18]
sk-2005	≈50.6M	≈1.8G	C	[WA]
uk-2007	≈106M	≈3.3G	C	[WA]
clueweb12	≈951M	≈37.3G	C	[WA]
uk-2014	≈787.8M	≈42.5G	C	[WA]
eu-2015	≈1.1G	≈80.5G	C	[WA]

2.3.3 Statistical Tests

In order to show that quality of partitions and running times of our algorithms differ from those of our competitors, we use statistical significant tests. In order to do this, we apply Wilcoxon signed-rank test [Wil45] to reject a null hypothesis that two sequences of random variables (measurements returned by two algorithms) have the same distribution. Specifically, the Wilcoxon signed-rank test returns the probability (p-value) to observe such sequences or even more different sequences under the null hypothesis. If p-value is less than 1% (significance level) then the null hypothesis is rejected otherwise it is not. To compare unpaired sequences of data, we use Mann-Whitney U test [MW47].

2.3.4 Machines

All experiments in this thesis were performed using the following three machines:

- Machine A has four Intel Xeon Gold 6138 processors (L1: 32 K, L2: 1024 K, L3: 28160 K, 4 sockets, 20 cores with Hyper-Threading, 160 threads) running at 2.0 – 3.7 GHz with 768 GB RAM and 4xSSD 1.8 TB.

- Machine B has two Intel Xeon E5-2683v2 processors (L1: 32 K, L2: 256 K, L3: 40960 K, 2 sockets, 16 cores with Hyper-Threading, 64 threads) running at 2.1 GHz with 512GB RAM and 1xSATA 447 GB.

- Machine *C* has two Intel Xeon E5-2650v2 processors (L1: 32 K, L2: 256 K, L3: 20480 K, 2 sockets, 8 cores with Hyper-Threading, 32 threads) running at 2.6 GHz with 128Gb RAM and 4xSSD 1 TB. (read 1440 MB/s, write 1440 MB/s).

All machines have Ubuntu 18.04 installed on them.

Chapter 3

3 Related Work

In this chapter, we give an overview of the graph partitioning techniques most of which are used within existing *parallel* graph partitioning frameworks. In our overview we focus on the graph partitioning techniques that can be parallelized efficiently. More comprehensive overviews of existing graph partitioning techniques can be found in [BS11; Bul+13; Fjä98; SKK03]. The keystone of almost all currently available parallel graph partitioning frameworks is the multi-level graph partitioning scheme (MGP) (see Section 3.1). The MGP scheme consists of three phases: coarsening, initial partitioning, and uncoarsening. Therefore, we present algorithms grouped by the phases they are used in. Note that our parallel graph partitioning techniques are described in the following chapters.

This chapter is organized in the following way. First, we discuss the details of the MGP scheme and describe why this heuristic approach is fast and constructs partitions of good quality in Section 3.1. In the following sections, we describe algorithms that can be used during every phase of the MGP scheme and their possible parallelizations. In Section 3.2, we give an overview of different clustering and matching algorithms that are used in the coarsening phase and are essential for resulting quality and running time. In Section 3.3, we review different parallel matching and clustering algorithms. Section 3.4 contains descriptions of different graph partitioning techniques that can be used in the initial partitioning phase. We concentrate on techniques that can be used within parallel graph partitioning frameworks. Therefore, we do not focus on other graph partitioning techniques like spectral graph partitioning [BS93; DH72; DH73; Fie73; Fie75; HL95b; XN98], tabu search [BH11; GBF11], simulated annealing [JS98], evolutionary/genetic algorithms [BH11; Kim+11], streaming graph partitioning [NU13; SK12; Tso+14] etc. Since we consider graphs without any additional geometrical information, we also do not present a overview of geometrical graph partitioning algorithms [FL93; Sim91; Wil91] and algorithms based on space-filling curves [Bad13; HW02; PB94; Zum12]. In Section 3.5, we consider different sequential refinement techniques that are used in the uncoarsening phase. Most of which are suitable for efficient parallelization. Section 3.6 contains descriptions of parallel refinement techniques. In Sections 3.7 and 3.8, we present a short overview of existing sequential and parallel MGP frameworks. Finally, Section 3.9 contains some

results on computational complexities of sequential approximation graph partitioning algorithms and on possibility of efficient parallelization of local search (one of the frequently used refinement techniques).

3.1 Multi-level Graph Partitioning

The multi-level graph partitioning (MGP) scheme is a widely used approach for graph partitioning. Most graph partitioning frameworks that do not construct optimal solutions employ the MGP scheme. The reason behind this is that the algorithms based on the MGP scheme often provide a good trade-off between running time and quality. The following paragraph gives a more detailed explanation of the advantages of the MGP scheme. The basic idea can be traced back to multigrid solvers for systems of linear equations. However, recent practical methods are mostly based on graph theoretic aspects. Specifically, the general idea of the MGP scheme is to recursively contract a graph until it is small enough to partition with a slow high-quality graph partition algorithm and, afterwards, uncontract the graph, while at the same time improving solution quality using refinement techniques. The MGP scheme consists of three main phases: coarsening, initial partitioning, and uncoarsening that are outlined in Figure 3.1.

The coarsening phase works in rounds and a hierarchy of coarse graphs is constructed in the end of the coarsening phase. During each round of the coarsening phase, a clustering (matching) of a graph is constructed and the graph is contracted according to this clustering producing a coarser graph. If the graph is not coarse enough, the coarsening phase proceeds to the next round where it contracts the coarsest graph even further. The coarsening phase stops when the coarser graph is small enough. In the initial partitioning phase, a slow high-quality graph partitioning algorithm is used to partition the coarsest graph. After receiving an initial partition of the coarsest graph, it is recursively uncontracted and the existing partition is projected onto it with an additional refinement (the uncoarsening phase). Note that each partition of the coarse graph corresponds to a partition of the finer graph. Although the main idea is simple, the MGP scheme has a lot of variations, since it is highly dependent on the following aspects: how is the original graph contracted, what algorithm is used to partition the coarsest graph, and what algorithm is used to refine the partition during the uncoarsening phase. Different variations of the MGP scheme were developed and analyzed independently by multiple researchers [Bou98; Gup97; HL95a; KK95b; KK98a; KK98c; MPD00; Pon+94; Pre01; Wal04; WC00a]. Note that we are interested in the MGP algorithms that have $\mathcal{O}((|E| + |V|)\mathsf{polylog}(|E| + |V|))$ running time since they yield a good trade-off between quality and running time.

Analyses of Multi-level Graph Partitioning. The multi-level graph partitioning (MGP) scheme is a heuristic that successfully combines global and local views on

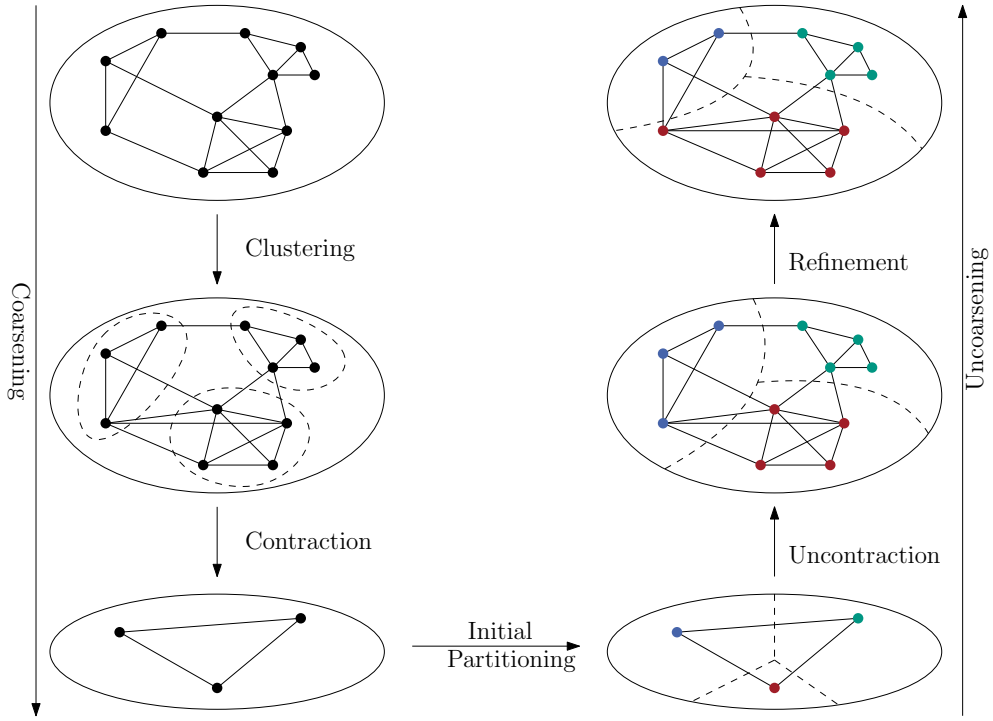


Figure 3.1: Outline of multi-level graph partitioning scheme.

the structure of the graph. As a result, it computes good quality partitions in reasonable time compared to other heuristics, approximation or exact algorithms. This is achieved by applying refinement techniques to each graph from the hierarchy of coarse graphs. Note that a vertex of a coarse graph corresponds to multiple vertices of the corresponding finer graph. Hence, a move of a vertex of the coarse graph corresponds to moves of multiple vertices of the finer graph. Therefore, refinement techniques applied to the coarser graphs perform moves of large number of vertices simultaneously which corresponds to the *global* view on the original graph, whereas refinement techniques applied to the finer graphs will perform moves of a small number of vertices which corresponds to the *local* view on the original graph. See the example in Figure 3.2.

Karypis and Kumar [KK95a] presented a theoretical analyses of the MGP scheme for bipartitioning using either the random matching or the heavy-edge matching algorithm in the coarsening phase (the latter one is described in Section 3.2.1). The authors consider 2D and 3D finite element meshes proving upper bounds on cut sizes for these graphs. To derive their bounds, the authors assume that the balanced vertex separator of a graph has a size of at most $\alpha|V|^{\gamma}$, where α and γ are graph-specific constants, as

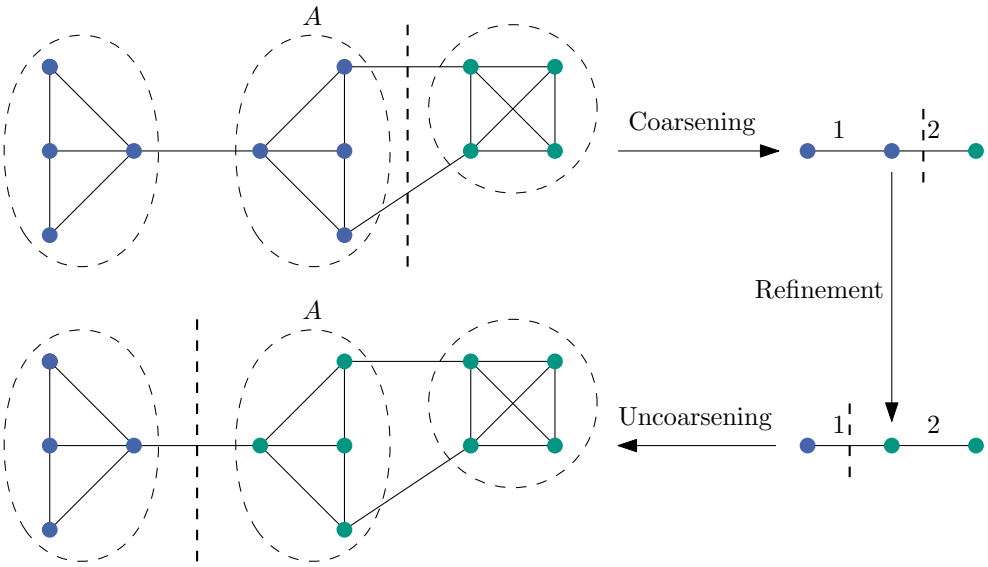


Figure 3.2: Moving the set of vertices A from the blue block to the green block decreases the cut size by one. But moving each vertex separately increases the cut size.

well as that the weights of edges and the degrees of vertices are uniformly distributed over the graph. Hence, cutting edges that are adjacent to the vertices of the vertex separator, we obtain a cut that has at most $\alpha(2|E|/|V|)(\sum_e w(e)/|E|)|V|^\gamma$ edges. Note that the bound does not hold if the average degree of vertices in the vertex separator is greater than the average degree of all vertices. However, using this assumption, the authors show that the size of any balanced bipartition of a coarse graph is $c \in [1, 2]$ times greater than the size of any balanced bipartition of the corresponding finer graph. On 2D and three $3D$ finite element meshes, the empirical analysis of the MGP scheme for bipartitioning confirms that the size of a bipartition of a coarse graph found by a heuristic algorithm is not much greater than the size of a bipartition of the finer graph found by the same heuristic algorithm. Furthermore, the authors prove an upper bound for the vertex separator in maximal planar graphs that is projected from a coarse graph to the corresponding fine graph. More precisely, the vertex separator of the finer graph is at most 1.5 times greater than that of the coarse graph. However, the measurements were performed only for one .

Walshaw [Wal03] investigates the effects of the coarsening phase on the size of the solution space (the set of all possible bipartitions) and which bipartitions remain in the solution space after coarsening. More precisely, the author shows that coarsening reduces the average size of a bipartition in the solution space by fixing some pairs of vertices to be in the same block. However, this reduction highly depends on the density

of a graph. If a graph is sparse then the number of high-cost solutions decreases faster than the number of low-cost solutions during the coarsening phase. But for graphs with high density, the number of high-cost solutions decreases approximately at the same rate as the number of the low-cost solutions. Therefore, refinement techniques, which search through in the solution space, will find a low-cost solution faster in a coarse graph constructed from a graph with low density. But coarsening will not help as much finding a low-cost solution for graphs with high density. This is confirmed by experimental comparison of the modified Fiduccia and Mattheyses local search algorithm (see Section 3.5.2) and the MGP algorithm with modified Fiduccia and Mattheyses local search in [Wal04]. The MGP algorithm finds smaller cuts than the modified Fiduccia and Mattheyses local search algorithm for graphs with low density. But for graphs with high density, both algorithms find cuts of approximately the same size. Although these experiments were performed on a small set of graphs – thus the results may differ on other types of graphs – they still give interesting insights on how a combination of coarsening and local search performs.

3.2 Coarsening

The coarsening phase consists of two main steps: constructing a clustering (matching) and contracting the graph according to the clustering (matching). The first step is the most important since the choice of a clustering (or a matching) algorithm directly affects the resulting quality and running time. The graph contraction is mostly straightforward and well-known; we describe it in Section 2.1.1. The parallelizations of the clustering and matching algorithms are described in Section 3.3.

3.2.1 Matching Based Coarsening

Matching based coarsening is a popular approach used in different MGP frameworks (see Section 3.7). The idea is to use a weighted matching algorithm to construct a matching which maximizes the sum of its edge weights. Then the corresponding graph is contracted according to the matching; that is, each matched edge and its incident vertices are contracted into one supervertex. The greater the weight of the matching is, the "better" the resulting coarse graph; that means it better preserves the structure of the finer graph.

There are a variety of optimal maximum weighted matching algorithms but their running times are super-linear. For example, the most recent optimal algorithm by Pettie [Pet12] runs in $\mathcal{O}(N|V|^{\omega})$ running time, where $\mathcal{O}(|V|^{\omega})$ is the running time of $|V| \times |V|$ matrix multiplication and N is the maximum edge weight in the graph. Since we are interested in a graph partitioning algorithm that has at most $\mathcal{O}((|E| + |V|)\mathsf{polylog}(|E| + |V|))$ running time, we use approximation matching algorithms with

the same order of complexity. However, note that these algorithms give only an approximation of the maximum weighted matching.

The first $\frac{1}{2}$-approximation algorithm for the maximum weighted matching was presented by Avis [Avi83]. This is a greedy algorithm that first sorts edges in decreasing order of their weights and then it iterates over these edges in order. Each edge is added to the matching if it is not incident to any of the edges in the matching. The running time of this algorithm is $\mathcal{O}(|E| \log |E|)$ since the sorting is the most time consuming part of the algorithm. Several attempts were made to improve the running time in the subsequent research.

Karypis and Kumar [KK95b] presented a linear time heuristic algorithm for the maximum weighted matching called the heavy-edge matching algorithm. This algorithm iterates over vertices choosing for each vertex the heaviest incident unmatched edge. Unfortunately, this algorithm does not have any quality guarantees since the algorithm may construct a matching with weight arbitrary less than the weight of the optimal solution. However, the algorithm is fast since it only iterates over all the edges once. The authors compare the heavy-edge matching algorithm with three other matching algorithms developed by them. In summary, the experiments indicate that the heavy-edge matching algorithm has the best trade-off between quality of the final partition and the running time, we do not describe the other algorithms.

Gupta [Gup97] presents the heavy-triangle matching algorithm. The idea is to match not two but three vertices by choosing a random vertex and two of its neighbors maximizing the sum of the weights of the edges incident to these three vertices. MGP algorithms with both matching algorithms show comparable quality of partitions and running times.

Preis [Pre99] presents the first linear time $\frac{1}{2}$-approximation algorithm for the maximum weighted matching called the local max algorithm. The main idea is to find a local max edge, add it to the matching, and remove all edges incident to it from consideration. An edge e is a local max if $\forall\ e'\ w(e) > w(e')$, where e' is a unmatched edge and there is an unmatched path between e and e'. The algorithm stops when there are no more edges to process. In summary, the running time of the algorithm is $\mathcal{O}(|V| + |E|)$.

Drake and Hougardy [DH03a] present another linear time $\frac{1}{2}$-approximation algorithm for the maximum weighted matching problem that is called the path growing algorithm. It is simpler to implement and analyze than the algorithm by Preis [Pre99]. The main idea of it is to grow a path from an arbitrary vertex alternatingly adding the heaviest unmatched edges to one of two different matchings. In the end, the algorithm returns the matching with the highest weight. The running time of the algorithm is $\mathcal{O}(|V| + |E|)$ since each edge is processed only once.

Drake and Hougardy [DH03b; VH05] present the first $(\frac{2}{3} - \epsilon)$-approximation algorithm. The algorithm tries to improve a maximal matching M by finding so-called β-augmentations and applying them to the matching, thus, increasing its weight. A β-augmentation A is a path or a cycle of an even length whose edges alternatingly either

belong or do not belong to M and $\beta \cdot \sum_{e \in M \cap A} w(e) < \sum_{e \in A \setminus M} w(e)$. Hence, setting $M := M \triangle A$ will increase the weight of M. The algorithm finds a corresponding β-augmentation of constant size for each edge of the initial matching M and applies it. Therefore, the running time of the algorithm is still $\mathcal{O}(|E|)$ assuming ϵ is constant. If ϵ is not constant then the running time of the algorithm is $\mathcal{O}(|E|\frac{1}{\epsilon})$.

Pettie and Sanders [PS04] presented two $(\frac{2}{3} - \epsilon)$-approximation algorithms for the maximum weighted matching problem. The first algorithm is a randomized algorithm that constructs a matching with an expected weight of at least $\frac{2}{3} - \epsilon$ times the weight of the optimal matching in $\mathcal{O}(|E| \log \frac{1}{\epsilon})$ expected time. The main idea of this algorithm is to choose a random vertex and augment the matching with an augmentation of constant size. The authors prove that choosing $\mathcal{O}(|V| \log \frac{1}{\epsilon})$ random vertices to improve the matching is enough to construct a matching with the desired expected weight. The second algorithm is a deterministic algorithm that always constructs a $(\frac{2}{3} - \epsilon)$-approximation of the optimal matching. The main idea here is to choose a set of augmentation and then to apply only a subset of them chosen in a greedy manner; that is, the algorithm repeatedly applies the augmentation with the maximum weight increase that is also edge and vertex disjoint with the previously selected augmentations. The worst-case running time of the algorithm is $\mathcal{O}(|E| \log \frac{1}{\epsilon})$. Additionally, the authors show how to produce a δ-approximation of the optimal matching for an arbitrary $\delta < 1$, however the running time becomes super-linear for $\delta \geq \frac{2}{3} - o(1)$. The advantages of these algorithms over the previous $(\frac{2}{3} - \epsilon)$-approximation algorithm by Drake and Hougardy [DH03b; VH05] are their simpler analysis and faster convergence to the $(\frac{2}{3} - \epsilon)$-approximation (exponential versus linear).

Drake and Hougardy [DH03c] compare four approximation algorithms for the maximum weighted matching problem: the greedy algorithm, the improved path growing algorithm, the local max algorithm, and the heavy-edge matching algorithm. Additionally, they apply a set of local improvements that further increases the weight of a matching. The authors prove that if there are no local improvements then the constructed matching is a $\frac{2}{3}$-approximation of the optimal matching. The authors measure running times and weights of the computed matchings for different classes of graphs. For most instances, the path growing algorithm constructs the best matchings. After applying local improvements, the differences between the computed weights and the optimal weights reduce by a factor of about two on average. The fastest algorithm is the heavy-edge matching algorithm while other algorithms are about 2 - 3 times slower on average.

Maue and Sanders [MS07] presented a new $\frac{1}{2}$-approximation algorithm for maximum weighted matching called the global path algorithm. It is a combination of the greedy algorithm, the path growing algorithm and dynamic programming. More precisely, the algorithm sorts edges in decreasing order, constructs paths or cycles of even length. Then it finds the maximum weighted matching in these subgraphs in linear time using dynamic programming. Additionally, the authors apply the random matching algorithm by Pettie and Sanders [PS04] to improve the matching. Therefore, in

expectation the algorithm constructs a $(\frac{2}{3} - \epsilon)$-approximation and its running time is proportional to the time to sort the edges. The authors compare the global path algorithm against the greedy algorithm, the improved path growing algorithm, and the random matching algorithm by Pettie and Sanders (applied to an empty matching). The authors measure running times and weights of the computed matchings for different classes of graphs. The global path algorithm constructs matchings with the smallest difference between their matching and the optimal matching on almost all instances, however the algorithm is the second slowest one.

There are other approximation algorithms for maximum weighted matchings that provide approximation ratios greater than $\frac{1}{2}$. Duan and Pettie [DP10] presented an algorithm that constructs a $(\frac{3}{4} - \epsilon)$-approximation in $\mathcal{O}(|E| \log |V| \log \frac{1}{\epsilon})$ time. Hanke and Hougardy [HH10] also presented a $\frac{3}{4}$-approximation algorithm that has $\mathcal{O}(|E| \log |V| \log \frac{1}{\epsilon})$ running time. The first $(1 - \epsilon)$-approximation algorithm that runs in $\mathcal{O}(|E| \frac{1}{\epsilon} \log \frac{1}{\epsilon})$ time was presented by Duan and Pettie [DP14].

Edge ratings. Several papers suggest to calculate maximal weighted matchings according to edge ratings instead of edge weights. Edge ratings incorporate information about how "well" two vertices of an edge are connected and, thus, how the matching of this edge will affect quality of the resulting partition. Holtgrewe et al. [HSS10] presented and analyzed four edge ratings. Furthermore, the authors use these four edge ratings in the `KaHIP` framework instead of edge weights in the matching algorithm. `KaHIP` that uses any of these edge ratings produces partitions of better quality than `KaHIP` that uses edge weights. Furthermore, partitions produced by `KaHIP` using these edge ratings have comparable quality. Osipov and Sanders [OS10] suggest the following edge rank $rank(u, v) = \frac{w(u,v)}{w(u) \cdot w(v)}$. Glantz et al. [GMS14] presented another edge rating. First a collection of minimum spanning trees is computed where the weight of an edge depends on how often the edge appears on different shortest paths. Each edge of a minimum spanning tree induces a cut in the original graph. We calculate the rating of an edge by considering the light cuts induced by it. The authors implemented this edge rating within the `KaHIP` framework and performed experiments on a large collection of graphs. Although this edge rating improves the maximum communication volume, it does not affect the cut size compared to other edge ratings. Safro et al. [SSS12] suggested another edge rating based on algebraic distance [CS11]. This distance distinguishes local and global edges. An edge is local if after removing it the distance between its endpoints remains small. The authors perform experiments on a large collection of graphs and show that the `KaHIP` framework produces smaller cuts with the new edge rating than with edge ratings by Holtgrewe et al. [HSS10].

3.2.2 Clustering Based Coarsening

In this section, we describe clustering algorithms that can be used in the coarsening phase. The common feature of these algorithms is that they allow clusters of size

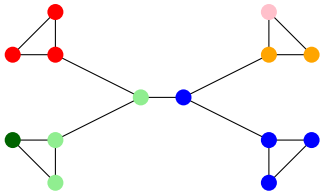
greater than two (a matching can be considered as a clustering where each cluster has size at most two). Contracting whole clusters allows to contract a graph faster then just contracting matched vertices. Clustering algorithms assign "well" connected vertices to the same cluster such that the structure of the coarsened graph is similar to the structure of the original graph.

Abou-Rjeili and Karypis [AK06] consider multiple extensions of matching based algorithms where incident edges are allowed to be matched. These extensions match irregular graphs more efficiently than common matching based algorithms. Specifically, a coarse graphs tend to be smaller and the connectivity seems to be preserved better. The authors consider two strategies to visit edges. Assume that we have a global order of edges. The first strategy visits edges in this order, whereas the second strategy chooses a random vertex and visits its incident edges in this order. Furthermore, the authors consider several ways to construct a global order of edges. They try to sort edges by different criteria: weight, the degree or weight of incident vertices, etc. In summary, the running times of matching algorithms are at most $\mathcal{O}(|E| \log |E|)$. The authors consider multiple combinations of the aforementioned strategies and edge orderings comparing them in pairs. Additionally, the authors compare MGP algorithms that use these strategies for coarsening to the `Metis` framework and the `Chaco` framework which both use matching based coarsening. On almost all graphs one of the MGP algorithms with clustering based coarsening produced better cut than that of `Metis` and `Chaco`. However, there is no strategy that produces better cuts on at least half of the graphs.

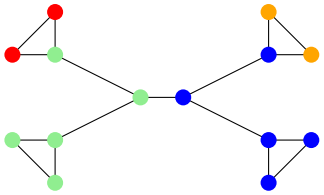
Meyerhenke et al. [MSS14] adapt a clustering algorithm called label propagation by Raghavan et al. [RAK07] for graph partitioning. The authors refer to the algorithm as label propagation with size constraints. The algorithm works in iterations. In the beginning, each vertex belongs to its own cluster. During an iteration, each vertex chooses a new cluster the one which most of its neighbors belong to; that is, the new cluster of a vertex v ($C[v]$) is

$$\arg\max_{c} \sum_{u \in N(v):\ C[v]=c} w(v,u).$$

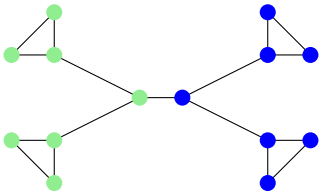
If there are several candidates then the tie is broken randomly. This additional randomness results in better partitions. Meyerhenke et al. modified the algorithm such that vertices change their cluster while respecting the size constraint. More specifically, a vertex chooses a new cluster only from the subset of neighboring clusters such that the size of the neighboring cluster does not exceed the size constraint if vertex moves there. Figure 3.3 shows the intermediate clusterings during three iterations of the label propagation algorithm. Several stopping rules are possible in the label propagation algorithm. The original stopping rule by Raghavan et al. [RAK07] signals to stop when each vertex can be moved only to clusters of the same size or smaller than its current cluster. Meyerhenke et al. suggest several ideas for the label propagation algorithm that improve its running time and quality of the resulting



(a) A possible clustering after first iteration.



(b) A possible clustering after second iteration.



(c) A possible clustering after third iteration.

Figure 3.3: These figures show possible clusterings produced by the label propagation algorithm. The label propagation algorithm considers vertices in increasing order of their degrees.

clustering. The first idea is to use an active vertex strategy. When using this strategy, the label propagation algorithm iterates only over active vertices. A vertex is active if at least one of its neighbors changed its cluster in the previous iteration; and all vertices are active in the beginning. Since the label propagation algorithm iterates only over active vertices, it mainly processes vertices on the border of clusters, the number of which tends to decrease over the course of the algorithm. Thus, the active vertex strategy significantly reduces the number of processed vertices in successive iterations. Additionally, Meyerhenke et al. use another stopping rule that signals to stop after a preset number of iterations or when all vertices are non-active. To improve quality of the clustering, the authors suggest to process vertices in increasing order of their degrees. The intuition behind this is that by processing low degree vertices first the algorithm tends to form clusters by adding low degree vertices to the clusters of high degree vertices. When the high degree vertices eventually are processed they are assigned to the clusters most of their low degree neighbors already belong to. The authors investigate other orderings, e.g., a weighted degree ordering, but these orderings do not improve quality of resulting clusterings. In summary, the running time of the algorithm is $\mathcal{O}(|V| \log |V| + t|E|)$, where t is the number of iterations. Meyerhenke et al. use this clustering algorithm during the coarsening phase of the `KaHIP` framework. The experiments indicate that the MGP algorithm with the label propagation algorithm produces partitions with an average cut size less than that of the `Metis` framework, the `Scotch` framework, and other configurations of `KaHIP`. It would be interesting to see additional experiments that investigate how the clustering and matching algorithms affect quality of partitions on different types of graphs.

3.3 Parallel Coarsening

In this section we consider different parallel clustering and matching algorithms that can potentially be used or actually used in parallel MGP frameworks.

3.3.1 Parallel Matching Based Coarsening

Karp et al. [KUW85] present a parallel optimal maximum weighted matching algorithm for graphs with edge weights polynomial in the number of vertices. We refer to such graphs as polynomial-weighted graphs. The main idea is to construct a special matrix whose determinant allows us to find the optimal weighted matching.

Chen and Uehara [UC00] present two approximation algorithms for the maximum weighted matching problem. The first algorithm is a randomized $(1-\epsilon)$-approximation algorithm that uses an optimal parallel maximum weighted matching algorithm for polynomial-weighted graphs by Karp et al. [KUW85] as a subroutine. The second algorithm is a parallelization of the greedy algorithm. Note that this algorithm computes a $(\frac{1}{2}-\epsilon)$-approximation, whereas the sequential greedy algorithm computes a $\frac{1}{2}$-approximation. The algorithm uses the parallel maximal matching algorithm by [IS86] as a subroutine. Both algorithms are theoretical since both subroutines need $\Omega(|E|+|V|)$ processors.

Hoepman [Hoe04] present a parallel $\frac{1}{2}$-approximation algorithm of the maximum weighted matching for the distributed memory model. This algorithm is a parallelization of the local max algorithm by Preis [Pre99] and works in rounds. During a round, each vertex sends a request along its edge with a maximum weight. If it receives back a positive answer along this edge then the edge is matched since it is a local maximum. If it receives a negative answer then the algorithm sends a request along the edge with the second largest weight and so on. Note that the running time of the algorithm is $\mathcal{O}(|E|)$ in the worst case.

Later Manne and Bisseling [MB07] modified the aforementioned algorithm for more practical use when the number of processors is less than the number of vertices and analyzed it for graphs with random edge weights. The analysis is based on reducing the problem to the maximal independent set problem since the parallel algorithm by Hoepman [Hoe04] is a variation of the parallel algorithm by Luby [Lub85] that solves the maximal independent set problem. The algorithm by Luby terminates after $\mathcal{O}(\log|E|)$ rounds in expectation given random edge weights. Furthermore, the authors implement the algorithm and show an experimental evaluation of the running time on two graphs. The algorithm seems to scale well with increasing number of PEs but there is a need for more comprehensive experiments.

Auer and Bisseling [AB11] present another parallel approximation algorithm for the maximum weighted matching problem in the shared-memory model and GPU settings

called red-blue matching algorithm. This algorithm is a parallel version of the heavy-edge matching algorithm. Therefore, the algorithm does not provide any quality guarantees and may construct a matching with the weight arbitrary less than the weight of the optimal matching. The main idea is to randomly color vertices in red or blue and then adjacent pairs of differently colored vertices decide if their edge belongs to the matching. More precisely, a vertex of one color proposes to match and the vertex of other color decides whether to accept this or another neighbor. The authors perform experiments on multiple graphs measuring running time and quality of the shared-memory and the GPU versions of the algorithm. Both algorithms scale well but their quality is worse than that of the greedy algorithm.

Birn et al. [Bir+13] present a parallel $\frac{1}{2}$-approximation algorithm called local max of the maximum weighted matching problem which is a parallelization of the local max algorithm. Although the parallel worst case running time is linear for weighted graphs, the algorithm has linear work and polylogarithmic parallel running time on graphs with unit edge weights. Furthermore, the experiments indicate that the algorithm shows good scalability. The authors consider a slightly different version of the sequential algorithm. It works in phases and during each phase it processes edges marking an edge if its weight is greater than the weight of any other incident edge and breaking ties randomly. Afterwards, all marked edges are added to the matching. The problem in the parallel algorithm appears during the search of a local max edge. Specifically, to find a local max edge in a chain with strictly increasing weights, the algorithm requires a number of phases equals to the length of the chain. The authors perform several types of experiments measuring the running time and quality of the sequential, the distributed memory, and the GPU versions of the algorithm. The sequential local max algorithm is compared to the global path algorithm, the greedy algorithm, the red-blue matching algorithm and the heavy-edge matching algorithm. The sequential algorithm shows almost the same quality as the global path algorithm that constructs matchings of better quality than other algorithms. Furthermore, the running times of the sequential algorithm are better than that of the global path algorithm. Both the distributed memory and the GPU versions of the algorithm show good scalability. Moreover, on large graphs the GPU version of the algorithm is faster than the GPU version of the red-blue matching algorithm.

LaSalle and Karypis [LK13] described a parallel shared-memory heavy-edge matching algorithm. The authors present three different approaches to parallelize the algorithm. In all approaches the vertices are assigned to PEs and each PE matches vertices assigned to it. The first approach is called *fine-grained matching*. The idea is to maintain a shared matching array M of size $|V|$ such that if vertices v and u are matched then $M[v] = u$ and $M[u] = v$. Then when a PE wants to match a vertex v with a vertex u ($v < u$) it locks $M[v]$ and $M[u]$ and checks if both vertices are not matched. Then if this is true it sets $M[v] = u$, $M[u] = v$ and releases both locks. To decrease the space consumption, the authors do not allocate locks for all vertices. Instead, they allocate a fixed number of locks that is significantly greater than p. Then to lock a vertex, the algorithm uses a hash function that maps a vertex to a

random lock. The second approach is called *multi-pass matching* and eliminates the use of locks. The idea is that each PE tries to match vertices assigned to it. If the PE matches two vertices and one of them belongs to other PE, it inserts this pair of vertices into a request buffer of the other PE. When all PEs finished to process their vertices, each PE processes the corresponding request buffers of other PEs and either accepts or rejects to match the pairs of vertices in the buffers. The algorithm repeats the aforementioned steps several times to increase the weight of the matching. The third approach is called *unprotected matching* and combines the two previous approaches. The algorithm maintains a shared matching array and matches vertices similar to the *fine-grained matching* approach but it does not use locks. Instead after all PEs finished to generate matchings, each PE checks if every local vertex v has a correct matching (i.e., $M[v] = u$ and $M[u] = v$), if this is not the case the PE sets $M[v] = v$. Note that if the number of vertices is much greater than the number of processors then it is unlikely that many incorrect matching occurs. There is a known problem which is that matchings of graphs with biased degree distributions often contain only a small fraction of edges. The reason is that only one edge incident to a high degree vertex can be matched. Therefore, the resulting contracted graph does not shrink significantly. To overcome this problem, LaSalle et al. [LaS+15] suggest a *two-hop matching* algorithm. The idea is that two vertices can be matched together even if they are not connected by an edge but also if they have a common neighbor. First, the algorithm uses a conventional matching algorithm. Next, it uses the two-hop matching method for vertices that have not been matched to further extend the matching. The authors implemented the aforementioned matching algorithms in the `Mt-Metis` framework. The experiments indicate that using the two-hop matching algorithm reduces the running time by a factor of 1.6 on average. Furthermore, quality of partitions improved by 3.2%.

Karypis and Kumar [KK99] as well as Walshaw et al. [WCE97] present a parallel distributed memory heavy-edge matching algorithm. This version of the algorithm is similar to the multi-pass matching approach of the parallel shared-memory heavy-edge matching algorithm by LaSalle and Karypis [LK13]. The difference is that Karypis and Kumar suggest to color a graph and matching vertices of the same color with their neighbors. Note that the shared-memory algorithm processes all the vertices in parallel. Matching vertices that belong to the same color avoids the following conflicts. Assume that one PE matches a vertex v with a vertex u that belongs to another PE which in turn matches u with some other vertex. As a result, v will not be matched with u. Using a coloring to process vertices does not completely solve all problems. Consider two vertices v and u that belong to the same color and are matched by different PEs. The problem occurs when they both match with the same vertex w that belongs to another color. To solve the conflict, PEs corresponding to v and u send requests to the PE that owns the vertex w. This PE decides which request to confirm and which to decline. Since the algorithm resembles the *multi-pass matching* approach of the parallel shared-memory *heavy-edge matching algorithm*, we do not describe the details of this algorithm here.

Karypis and Kumar [KK97] present another parallel distributed memory *heavy-edge matching algorithm*. It is a distributed memory version of the *multi-pass matching* approach of the parallel shared-memory *heavy-edge matching algorithm* by LaSalle and Karypis [LK13]. We do not describe the details of the algorithm here.

Holtgrewe et al. [HSS10] present another parallel distributed memory approximation algorithm for the maximum weighted matching problem. First, the vertices are distributed over PEs using the initial numbering of the vertices. Next, each PE preforms the sequential *global path* algorithm to match its local vertices. Then the parallel distributed memory algorithm by Manne and Bisseling [MB07] is used to match the gap graph. The gap graph consists of local max edges (v, u) such that v and u belong to different PEs. Finally, the matching of the gap graph and the matchings constructed by all PEs are combined.

Her and Pellegrini [HP10] present a parallel distributed memory approximation algorithm for the maximum weighted matching problem. This algorithm is a modification of the algorithm by Karypis and Kumar [KK97]. The authors try to solve the following problem. A requested vertex v can either match with another local vertex or send a request itself. In the former case, the request is declined and the sender PE receives a notification. When the request of a PE is declined, it does not try to match with the requested vertex anymore. On the contrary in the latter case, the request is not replied because it is unclear whether the request can be satisfied or not. Since the vertices are usually distributed randomly across PEs, the probability that the neighbor connected by the heaviest edge is on another PE is high. Therefore, we should not only match local vertices, since preferring locality over maximizing edge weights deteriorates quality of the resulting matching. To find a trade-off between the number of local and non-local matched edges, the authors suggest to send a matching request with a probability 0.5. This randomized approach decreases the number of unanswered requests by preferring local matchings but still matches vertices from different PEs if it is advantageous.

Halappanavar et al. [Hal+12] suggest an improvement of the parallel distributed memory $\frac{1}{2}$-approximation algorithm of the maximum weighted matching by Manne and Bisseling [MB07]. The sequential algorithm finds a candidate to match with for each vertex. If the candidate also matches with the chosen vertex then both vertices are inserted into a queue. After constructing the queue of matched vertices, the candidates of the remaining vertices must be updated. Specifically, the algorithm extracts a matched vertex v from the queue and for each its neighbor that has v as a candidate finds a new candidate. If a neighbor matches with some vertex then both vertices are inserted into the queue. The algorithm stops when the queue is empty. The parallelization of the algorithm is straightforward and works in iterations. Each iteration consists of two phases. In the first phase, the parallel algorithm finds candidates for matching for all vertices. Then it iterates over all vertices again and inserts all pairs of matched vertices into a queue. During the second phase, the algorithm extracts vertices from the queue finding new candidates for their neighbors.

If a neighbor matches with a vertex then both vertices are inserted into the queue for the next iteration. If the queue for the next iteration is empty then the algorithm stops. Otherwise, it swaps the queues and proceeds with the next iteration. The authors present an experimental evaluation of their parallel algorithm on different machines. The algorithm scales well (close to linear). However, the authors evaluate their algorithm only on three classes of synthetic graphs. It would be interesting to see experiments on a larger benchmark set.

Manne and Halappanavar [MH14] present a parallel shared-memory $\frac{1}{2}$-approximation algorithm of the maximum weighted matching problem called Suitor. The algorithm has an advantage over the parallel algorithm by Halappanavar et al. [Hal+12] since it does not use concurrent queues. The authors prove that their algorithm constructs the same matching as the *greedy algorithm*. The idea is to find local max edges by computing suitors for all vertices. A vertex v is a new suitor of a vertex u if $w(v,u)$ is greater than the weight of the edge between u and its current suitor and u maximizes $w(v,u)$. If we computed a new suitor v of a vertex u and u already has a suitor w then u is assigned the new suitor v and we need to find a new vertex for which w can be a suitor. Thus, we recompute a suitor for a vertex v at most $d(v)$ times which gives the total running time complexity $\mathcal{O}(|E|\Delta)$. The parallelization of the *Suitor* algorithm is straightforward: All PEs process vertices in parallel computing suitors for them. The *Suitor* algorithm shows better running times compared to the *local max algorithm*. Note that the authors compare their algorithm against the implementation of the *local max algorithm* that uses locks. However, it is possible to implement the algorithm without using locks. In practice, matchings constructed by *greedy algorithm* have smaller weights than that of constructed by the *path growing* and *global path* algorithms. Thus, the authors suggest an improvement of their algorithm. The idea is to construct two disjoint matchings of a graph and to combine them using dynamic programming producing the maximum weighted matching for the edges of both matchings. The authors compare their algorithm to different sequential algorithms. The experiments indicate that the improved *Suitor* algorithm and *global path* algorithm produce matchings of comparable quality. In terms of the running time, the *Suitor* algorithm outperforms all its competitors including the parallel algorithm by Halappanavar et al. [Hal+12].

3.3.2 Parallel Cluster Based Coarsening

Narang and Soman [SN11] presented a parallel clustering algorithm that optimizes the modularity measure [New06]. It is based on the label propagation algorithm. They propose several ways how to improve quality of the label propagation algorithm: computing weights of edges according to their "connectivity importance" and assigning an identical label to strongly connected vertices. The authors probably use a parallel for loop with internal scheduler of the used library. Furthermore, the authors measure

the scalability of their clustering algorithm. They do not show measurements for the parallel label propagation algorithm alone.

Fagginger Auer and Bisseling [AB12] present a parallel matching algorithm that is used as a subroutine in their multi-level clustering algorithm which optimizes the modularity measure. The idea is to contract a graph until it is small enough to apply a suitable clustering algorithm. An important detail of this matching algorithm is that it matches star-like vertices and their neighbors (that is, matching may contain more than two vertices). Here, a vertex is star-like if it has many neighbors of small degree. This approach allows to overcome the known problem of using matching algorithms in MGP algorithms: a common matching algorithm matches a star-like vertex only with one of its neighbors leaving other neighbors potentially unmatched. Thus, the size of a resulting coarse graph does not reduce sufficiently and it takes more time to contract the input graph until it is small enough. The experiments indicate that the algorithm has good running time and quality.

Meyerhenke and Staudt [SM16] present a parallel label propagation algorithm that is used as a subroutine in their multi-level clustering algorithm which optimizes the modularity measure. The idea is to contract a graph until it is small enough to apply a suitable clustering algorithm. First, the algorithm constructs a clustering of a graph using the parallel label propagation algorithm, which optimizes modularity, and contracts the graph according to the clustering. Next, it recursively applies the parallel label propagation algorithm to the contracted graph until the clustering of the coarsest graph does not change. Then it propagates the clustering to the original graph. In the parallel label propagation algorithm, each PE iterates over a range of vertices and chooses a new cluster for each vertex. The authors use the "parallel for" implementation of the *OpenMP* library [DM98] using a guided scheduler to parallelize the processing of vertices.

Duriakova et al. [Dur+14] analyze the changes in running time and quality of parallel label propagation algorithms that optimize modularity depending on the synchronization strategy of cluster IDs: synchronous, asynchronous, and semi-synchronous. The first strategy uses cluster IDs of neighbors according to the previous iteration to compute new cluster IDs of vertices. The disadvantage of this strategy is that each PE uses old cluster IDs; that is, it does not observe the changes made by other PEs during the current iteration and as a result the label propagation algorithm converges more slowly. The second strategy uses cluster IDs of neighbors in real-time; that is, cluster IDs are stored in a shared array. PEs read and write cluster IDs from/to this array, thus, data races are possible and it is impossible to predict if a vertex has its cluster ID according to the previous or current iteration. Furthermore, reads and writes to the same cache line of the array cause false sharing. The third strategy attempts to eliminate the aforementioned disadvantages such that each PE observes cluster IDs of neighbors according to the current iteration. For this, the authors proposed to color a graph and perform label propagation for vertices of one color in parallel color after color. All PEs update cluster IDs of the vertices that belong to the same color and,

thus, the label propagation algorithm observes the changes of cluster IDs during the iteration. But this method has its own disadvantages. One of them is the additional synchronization between processings two colors. The authors additionally analyze the impact of vertex ordering on asynchronous and semi-synchronous strategies. The experiments indicate that the semi-synchronous strategy has better running times and quality for three graphs. However, it would be interesting to see experiments on a larger benchmark since both strategies were tested only on three graphs and running times of the strategies can considerably fluctuate from graph to graph. Furthermore, it is not clear why the speed-up of the asynchronous strategy for eight PEs is much smaller than the speed-up of the parallel label propagation algorithm by Meyerhenke and Staudt [SM16] for the same graph. The authors also use the "parallel for" implementation of the *OpenMP* library [DM98] with a guided scheduler to parallelize the processing of vertices.

Khlopotine et al. [KSJ15] present a parallel label propagation algorithm that is used as a clustering algorithm which optimizes modularity. The authors optimize the label propagation algorithm for the Intel Xeon Phi architecture and use the "parallel for" implementation of the *OpenMP* library [DM98] with a guided scheduler to parallelize the processing of vertices. Their implementation shows the speed-ups similar to those of the implementations by Meyerhenke and Staudt [SM16] and Duriakova et al. [Dur+14].

Meyerhenke et al. [MSS17] present a parallel distributed memory label propagation algorithm. They use it to compute size-constrained clusterings in the coarsening phase and to refine partitions in the uncoarsening phases of their MGP algorithm. The idea is to distribute subgraphs induced by ranges of vertices over the PEs. More precisely, each PE locally has a subgraph induced by a range of vertices. Additionally, each PE has a set of vertices called ghost vertices. These are vertices that are adjacent to at least one vertex on the PE but reside on other PEs (adjacent PEs). One iteration of the distributed memory label propagation works similarly to that of the sequential label propagation algorithm. Each PE performs the label propagation algorithm on its subgraph selecting a new cluster ID for each vertex. To communicate the changes of cluster IDs between PEs, the algorithm uses the following approach. Each PE maintains a send buffer for each adjacent PE. If a vertex adjacent to a ghost vertex changes its cluster ID then the algorithm inserts the vertex and its new cluster ID to the send buffer that corresponds to the PE where the ghost vertex is stored. When the PE has found new cluster IDs for all their vertices and one iteration is over, it sends the changes stored in the send buffers to the adjacent PEs and receives the changes of cluster IDs of ghost vertices from them. The authors note that when the label propagation algorithm is close to its convergence, the communication volume is low since the number of vertices that change their cluster IDs is very small. To guarantee that the total weight of vertices in each cluster does not exceed the threshold $(1+\epsilon)|V|/(k \cdot f)$, two different approaches are used during the coarsening and uncoarsening phases. Here f is a parameter to control the trade-off between the speed of contraction of the input graph and the resulting quality of the partition. Specifically,

if f is large then the maximum weight of clusters is small and the coarse graph does not shrink fast. On the other hand, if f is small then the input graph is contracted too aggressively which can potentially result in a poor quality of the partition. We describe the approach used during the coarsening phase. In the beginning of the label propagation algorithm, each vertex is in its own cluster. For each cluster each PE maintains the total weight of its local vertices within that cluster and uses this information to prevent clusters from being overloaded. Each time a PE moves a vertex to a new cluster, it updates the local weights of the old and new clusters. Note that this approach does not require any additional communication. However, it is possible that some clusters may be overloaded since several PEs may independently move vertices to one cluster without knowing its actual total weight. The authors note that the threshold $(1 + \epsilon)|V|/(k \cdot f)$ is not tight during coarsening and, thus, some clusters may be overloaded.

3.4 Initial Partitioning

After the coarsest graph is small enough, the initial partitionings begins. Most of graph partitioning frameworks partition the coarsest graph several times with different random seeds. The best partition is selected in the end. This approach improves the resulting partition. The parallelization of this approach is straightforward; that is, each PE performs the partition of the coarsest graph with a random seed. After all PEs finish, the best partition among all PEs is selected. Note that in this section we consider different initial partitioning algorithms that can potentially be used in an MGP algorithm or that are actually used in parallel graph partitioning frameworks.

3.4.1 Exact Algorithms

In this section we describe only one exact bipartitioning algorithm since it the only one that is able to partition large graphs with tens of thousands of vertices and edges. Therefore, this algorithm is interesting from a practical point of view since it can be used in combination with recursive bisection (see Section 3.4.3) during the initial partitioning phase to compute a k-way partition. A detailed overview of other exact algorithms can be found in [Bul+13].

Delling et al. [Del+12] present a branch-and-bound algorithm that solves the bipartitioning problem. The first component of the algorithm is a novel technique to compute lower bounds of bipartitions. First, the author find a max-flow that is a lower bound for a bipartition. Next, they use a novel packing technique to improve the lower bound (the max-flow lower bound does not take into account the balance). The main idea of the packing technique is to consider a partial bipartition of the graph and to construct a special collection of subtrees called tree packing. They prove that it is possible to increase the lower bound by the number of trees in the tree packing.

The second component of the algorithm attempts to decrease the search space by considering several ways to extend a partial partition without explicit branching. Specifically, if adding a vertex to one of the blocks increases the lower bound over the upper bound then it is enough to consider the case when the vertex belongs to the other block.

The third component of the algorithm partitions edges into $U + 1$ disjoint sets of edges where U is the upper bound. The authors prove that at least one of these sets does not contain cut edges and, thus, contracting it does not affect the solution but reduces the size of the graph. The authors perform an extensive comparison of their algorithm and other exact bipartitioning algorithms: the quadratic programming based algorithm by Hager et al. [HPZ13], BiqMac [RRW10] (semidefinite programming), and SEN [KRC00] (multi-commodity flows). In summary, their algorithm partitions many graphs orders of magnitudes faster than their competitors. Furthermore, they optimally partition several large graphs from the DIMACS benchmark [Bad+13b] constructing optimal bipartitions for some graphs for the first time.

Note that if we want to perform initial partitioning using this algorithm then the original graph must be contracted even further to partition the graph in a reasonable time. For example, this algorithm partitions a random geometric graph with 2^{16} vertices in 14K seconds, whereas heuristics partition it much faster. However, additional contraction of the graph can decrease resulting quality of partitions. It would be interesting to investigate how the initial partitioning with the exact algorithm affects quality of partitions.

3.4.2 Graph Growing Partitioning

The graph growing partitioning [GL81; GS94; JL96; Sim91] is a fast and simple approach to bipartition a graph. The idea is to run a BFS from a random source vertex adding the first $\frac{|V|}{2}$ visited vertices to the first block and the rest $\frac{|V|}{2}$ vertices to the second block. However, the choice of a good source vertex affects the quality of the resulting bipartition. Thus, George and Liu [GL81] suggest to choose a so-called pseudo-peripheral vertex as a source vertex. To find such a vertex, the authors run a BFS from a random vertex v to find the most distant vertex u from v. Analogously, the most distant vertex from u is found and so on. The algorithm stops when the distances between the most distant vertices of the last two repetitions remain the same. Finally, one of the two vertices from the last repetition is selected as a source vertex.

Karypis and Kumar [KK98a] suggest a modification of the aforementioned algorithm by changing the way a BFS selects the next vertex to visit. The authors call this modification Greedy Graph Growing Partitioning. Instead of using a queue, the authors use a priority queue with gain as priority. Thus, the BFS visits a vertex with

the strongest connection to previously visited vertices. As a result, this modification decreases the cut size of the resulting bipartition.

Bubble Framework. Diekmann et al. [Die+00] suggest a generalization of the aforementioned algorithm for arbitrary values of k. Furthermore, they iteratively improve the partition by reapplying the algorithm several times using an improved set of source vertices.

The algorithm consists of three main phases: the selection of the initial source vertices, the construction of the blocks, and the selection of "better" source vertices. The algorithm selects initial source vertices as follows. It picks a random vertex and selects the most distant vertex from it as the first source vertex. Next, it selects the most distant vertex from the first source as the second source and so on until k source vertices are selected. After all blocks are constructed, the third phase starts. The algorithm constructs blocks by performing a BFS from the source vertices. It adds a new vertex to the lightest block given that the vertex is connected to the block. If such a block is connected to several vertices then the algorithm selects the vertex with the maximum sum of weights of edges that connect it to the block. In the third phase, the algorithm selects a new source vertex for each block such that the sum of the distances to the block's vertices is minimized. Unfortunately, it takes $\mathcal{O}(b^2)$ time to find such a vertex, where b is the size of the block. Therefore, the authors suggest another way to select new source vertices. They calculate the distances from the previous source vertex and its neighbors to all the remaining vertices and select the neighbor that minimizes the aforementioned value. This is repeated until a local minimum is found.

The complete algorithm works as follows. First, it performs the first phase to select initial source vertices. Next, it performs the second and the third phases until the cut size does not decrease for ten consecutive iterations or new source vertices remain the same.

3.4.3 Recursive Bisection

The recursive bisectioning technique [BB87; ST97] can be used to partition a graph into k blocks using a bipartitioning algorithm. The idea is to perform the k-way partitioning in a recursive manner. Suppose we want to partition a graph into k blocks and for simplicity assume that k is a power of two. Then we partition the graph into two blocks and recursively partition these two blocks into $k/2$ blocks each. Unfortunately, recursive bisection can construct a low-quality k-way partition even if an optimal bipartitioning algorithm is used. Simon and Teng [ST97] showed that recursive bisection can construct a k-way partition that is $\mathcal{O}(\frac{|V|^2}{k^2})$ times worse than the optimal partition of a dense graph with $\Theta(|V|^2)$ edges. Furthermore, the optimal partition of a sparse graph with $\Theta(\frac{|V|}{k})$ vertices can be $\mathcal{O}(\frac{|V|}{k})$ times smaller than

the partition constructed using recursive bisectioning. This is due to the fact that recursive bisection has a greedy nature and does not use global information about the graph. Therefore selecting an optimal bipartition can make it impossible to find small bipartitions in the following recursive bisections.

3.5 Refinement Techniques

After constructing the initial partition of the graph, it can be improved using refinement techniques. The most popular technique is a local search based on local moves of vertices between blocks. Although the greedy variation of this technique finds only a local minimum, there are approaches that can escape local minima by allowing moves with negative gains and subsequent rollback to the best found solution. These approaches us allow to move dense regions of graphs; each vertex in a dense region has a negative gain but moving the region together is beneficial. In the following section, we mostly consider different variations of local search since their parallelizations are used in almost all parallel graph partitioning frameworks. Section 3.6 gives an overview these parallelizations.

3.5.1 The Kernighan-Lin Local Search

Kernighan and Lin [KL70] present a local search heuristic that approximately solves the bipartitioning problem. The main idea is to exchange pairs of vertices gradually decreasing the cut size and preserving the balance of the partition. Consider a perfectly balanced bipartition of a graph, and let A^* and B^* be an *optimal* perfectly balanced bipartition of the graph. Then there are sets $X \subseteq A, Y \subseteq B$ ($|X| = |Y|$) such that after exchanging them between A and B (i.e., $A := (A \setminus X) \cup Y, B := (B \setminus Y) \cup Y$) the resulting bipartition is optimal. Unfortunately, the problem of finding theses sets is NP-hard. Therefore, the local search algorithm by Kernighan and Lin [KL70] exchanges pairs of vertices between blocks A and B trying to find a "good" approximation of X and Y. Obviously the choice of a pair to exchange is the most important part of the algorithm. To decide which pair to move, the authors assign a *gain* to each exchange. Specifically, the gain of an exchange of two vertices v, u between blocks A and B is $\text{gain}(v, u) := g(v) + g(u) - 2 \cdot w(v, u)$, where $g(v) := \sum_{a \in A} w(v, a) - \sum_{b \in B} w(v, b)$ is the gain of moving a vertex v from the block B to the block A. See Figure 3.4 for an example. Namely, the gain is a decrease of the cut size after exchanging a pair of vertices. If vertices are strongly connected to the vertices of the opposite block then exchanging them can be advantageous given the weight of the edge between them is relatively small.

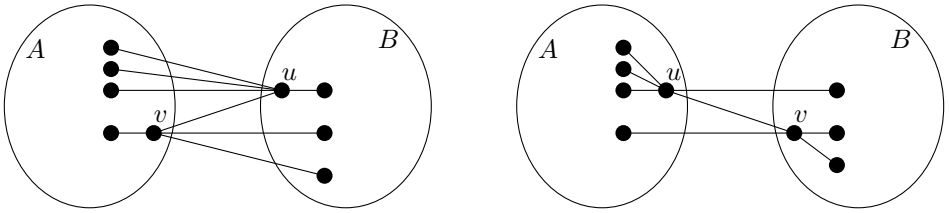


Figure 3.4: Exchanging vertices v and w will decrease the cut size by 3. Specifically, $g(v,u) = g(v) + g(u) - 2 \cdot w(v,u) = 3$, where $g(v) = 2$ and $g(u) = 3$.

Algorithm Description. The algorithm performs a preset number of passes. During a pass, it tries to find a sequence of vertex exchanges with maximum total gain. In order to do this, the algorithm selects a pair of vertices with the maximum gain (i.e. $\{v,w\} = \arg\max_{\{x,y\} \in A \times B} \text{gain}(x,y)$) and exchange them between blocks A and B. Both vertices are marked such that they cannot be moved during the current pass. Afterwards, the algorithm updates the gains of the neighboring vertices and selects new pair from the remaining vertices. The algorithm stores the best sequence of exchanges that results in the smallest encountered cut size. After $p = \min(|A|, |B|)$ vertices were moved, the algorithm backtracks to the partition with the smallest cut size and the pass is finished. Consider a sequence of exchanges $\{(a_i, b_i)\}_{i=1}^{p}$. Then the algorithm backtracks to the bipartition after first $\arg\max_{k \in \{1 \dots p\}} \sum_{i=1}^{k} \text{gain}(a_i, b_i)$ exchanges.

Pair Selection. The running time of local search depends on how vertex pairs are selected. The original algorithm proposed by Kernighan and Lin [KL70] first sorts the vertices of each block in decreasing order of their gains resulting in the two following sequences:

$$g(a_{i_1}) \geq g(a_{i_2}) \geq \cdots \geq g(a_{i_{|A|}}), \forall\ i_t \in \{1 \dots |A|\} \tag{3.1}$$

$$g(b_{i_1}) \geq g(b_{i_2}) \geq \cdots \geq g(b_{i_{|B|}}), \forall\ i_t \in \{1 \dots |B|\}. \tag{3.2}$$

Then it iterates over both sequences selecting a pair of vertices with maximum gain. Although in the worst case all possible pairs have to be scanned, it is very likely that the algorithm will find the pair with maximum gain quickly. This is due to the fact that if a pair of vertices (a_i, b_j) has a gain not greater than the maximum gain seen so far then all the following pairs (i.e., $(a_{i'}, b_{j'}), i' > i, j' > j$) will also have a gain that is not greater than the maximum gain. The experiments performed by the authors confirm that the sorting of gains dominates the selection of a pair with maximum gain. Thus the selection can be done in $\sum_{i=0}^{|V|-1} (|V| - i) \log(|V| - i) = \mathcal{O}(|V|^2 \log |V|)$ time. After exchanging a pair, the algorithm updates the gains of the remaining vertices in $\mathcal{O}(|V|)$ time by considering all remaining vertices and updating the gain of each in constant time. Therefore, the resulting running time of a pass is $\mathcal{O}(|V|^2 \log |V|)$ assuming that the selection of a pair with the maximum gain from the sorted list

takes $\mathcal{O}(|V|)$ time. However, without this assumption the worst case running time is $\mathcal{O}(|V|^3)$.

Another approach to select the pair with the maximum gain and update the gains of the remaining vertices was proposed by Dutt [Dut93]. First, Dutt shows that it is enough to scan only $\mathcal{O}(\Delta^2)$ pairs to find the pair with the maximum gain. The key idea is that once a pair of vertices without an edge between them is found, all the following pairs will have less or equal gain. Furthermore, the author uses balanced search trees to update the gains of vertices. The resulting running time of a pass is $\mathcal{O}(|V|\Delta^2 + |V|\Delta \log |V|)$.

Theorem 3.1

Given two sequences of sorted vertex gains

$$g(a_{i_1}) \geq g(a_{i_2}) \geq \cdots \geq g(a_{i_{|A|}}), \forall\ i_t \in \{1 \ldots |A|\} \tag{3.3}$$

$$g(b_{i_1}) \geq g(b_{i_2}) \geq \cdots \geq g(b_{i_{|B|}}), \forall\ i_t \in \{1 \ldots |B|\} \tag{3.4}$$

then it is enough to scan $\mathcal{O}(\Delta^2)$ pairs to find the pair with maximum gain.

Proof. Consider the first vertex a_{i_l} that is not connected to b_{i_1}. Then $\text{gain}(a_{i_l}, b_{i_1}) = g(a_{i_l}) + g(b_{i_1}) \geq g(a_{i_j}) + g(b_{i_k}) \geq \text{gain}(a_{i_j}, b_{i_k})$ for $j > l$ and $k > 1$.

Now consider the first vertex b_{i_t} that is not connected to a_{i_1}. Then $\text{gain}(a_{i_1}, b_{i_t}) \geq \text{gain}(a_{i_j}, b_{i_k})$ for $j > 1$ and $k > t$.

Thus, it is enough to scan all pairs a_{i_j}, b_{i_k}, where $j \leq l$ and $k \leq t$. Both l and k are $\mathcal{O}(\Delta)$. Hence, it is enough to scan only $\mathcal{O}(\Delta^2)$ pairs to find the pair with maximum gain. □

Extensions. Kernighan and Lin [KL70] present several extensions of their algorithm that are able to refine a bipartition with blocks of unequal size, to refine a partition given a graph with integer vertex weights, and to refine a k-way partition.

Consider partitioning a graph into blocks A and B such that $|A| \leq |B|$. Then, given the initial partition of the graph into A and B ($|A| < |B|$), the algorithm stops to exchange vertices as soon as $|A|$ pairs have been exchanged. Hence, the sizes of the blocks always remain the same. Furthermore, the authors consider another variant of the graph bipartitioning problem when each block must have at least $|A|$ and at most $|B|$ vertices. Then $|B| - |A|$ dummy elements are added to the block $|A|$ that are not connected to other elements. Afterwards, the algorithm performs passes and removes the dummy elements in the end. Note that after each pass both blocks have at least $|A|$ and at most $|B|$ real elements.

To solve the k-way partitioning problem, the authors suggest to perform refinement pairwise; that is, in each step a pair of blocks is selected and refined. Afterwards, the algorithm selects a new pair where at least one of the blocks has changed or has not

been refined before. Unfortunately, this approach has worse running time since at least $\binom{k}{2}$ pairs of blocks should be considered.

To solve the bipartitioning problem of a graph with integer vertex weights, the authors suggest to substitute a vertex with weight w by a clique of size w with significantly high edge weights. This prevents vertices of the same clique to be in different blocks. Unfortunately, this approach can drastically increase the size of the problem if the vertex weights are large.

3.5.2 The Fiduccia-Mattheyses Local Search

Fiduccia and Mattheyses [FM82] presented the modification of the Kernighan-Lin local search algorithm that approximately solves the graph bipartitioning problem on graphs with unit edge weights. The original description of the algorithm in the paper is for hypergraphs. The authors improved the running time of the Kernighan-Lin algorithm from $\mathcal{O}(|V|^2 \log |V|)$ to $\mathcal{O}(|E|)$. The main idea is to use a bucket priority queue to find the vertex with maximum gain and to update only gains of the vertices that are adjacent to this vertex. This algorithm or its modifications are used in most graph partitioning frameworks to refine partitions of graphs.

Algorithm Description. The algorithm works similarly to the Kernighan-Lin local search algorithm and performs a preset number of passes. During a pass, the algorithm maintains a bucket priority queue (which is described further) for each block that returns the vertex with maximum gain. Each bucket priority queue stores key-value pairs where the value is a vertex and the key is the gain of the vertex. Depending on imbalance allowance, two strategies for selection of a vertex for the next move are possible. When no imbalance is allowed then the algorithm alternatively extracts vertices from both bucket priority queues. On the other hand, if ϵ-imbalance is allowed then the algorithm considers two vertices with maximum gain from both priority queues. If the move of any of the vertices violates the balance constraint it is excluded from the following consideration. If the moves of both vertices do not violate the balance constraint then the algorithm selects the vertex with the largest gain. Furthermore, it breaks ties by selecting the vertex whose movement improves the balance better. Finally, the algorithm moves the selected vertex to the opposite block and updates the gains of its neighbors. The pass ends when both priority queues are empty.

Bucket Priority Queue. A bucket priority queue stores key-value pairs and returns a value with the maximum key in $\mathcal{O}(d_{\text{avg}}) = \mathcal{O}(|E|/|V|)$ amortized time as well as inserts/updates the key of a value in $\mathcal{O}(1)$ worse-case time. The data structure is based on the fact that edge weights are unit and, thus, gains are integers in the range $[-\Delta, \Delta]$. Therefore, it is possible to have an array of size $2\Delta + 1$ where each element

is a bucket implemented using a linked list. Each bucket stores vertices with equal gains. Figure 3.5 shows a bucket priority queue.

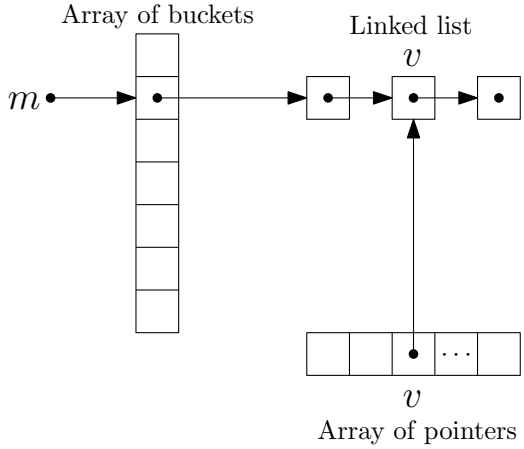


Figure 3.5: A bucket priority queue consists of two arrays: an array of buckets and an array of pointers. The first arrays contains buckets of vertices. The second array contains pointers to vertices in the buckets.

To insert a pair $(gain, v)$ to a bucket priority queue, one accesses the $gain + \Delta$-th bucket in the array of buckets and inserts the vertex v into that bucket. To perform an update operation, an additional array of size $\mathcal{O}(|V|)$ is necessary that contains pointers to vertices in the bucket priority queue. More precisely, an i-th element of the array is a pointer to the vertex i in the bucket priority queue. Then to update the gain of a vertex v to value g, one accesses the v-th pointer in the array of buckets, removes the vertex v from the corresponding bucket, and reinserts it in the $g + \Delta$th bucket. Since an insert/update operation performs only a constant number of memory accesses and an insert/delete operation of a linked list takes $\mathcal{O}(1)$ time, the running time of an insert/update operation of a bucket priority queue is $\mathcal{O}(1)$.

To perform an extraction operation of a vertex with the maximum key, each bucket priority queue additionally maintains an index m of the bucket with maximum gain. Then an extraction operation deletes a vertex from the m-th bucket and returns it. Now if that bucket becomes empty, the index m must be updated by decreasing m until the corresponding bucket is not empty. The following theorem shows that the amortized number of such decrements is $\mathcal{O}(d_{\text{avg}})$. We present this theorem since it is not discussed in detail in the literature about bucket priority queues. Furthermore, this bucket priority queue differ from the bucket priority queue used in the shortest path algorithms [MS08] which inserts keys only in increasing or decreasing order.

Theorem 3.2
The total cost of $|V|$ update operations is

$$\mathcal{O}(\sum_{v \in V} d(v)) = \mathcal{O}(|E|).$$

Therefore, the amortized cost of an update operation is $\mathcal{O}(\frac{|E|}{|V|}) = \mathcal{O}(d_{avg})$.

Proof. Consider an update operation that extracts a vertex v and then searches for a new vertex u with maximum gain. And let us denote this vertex as u. Then in the worst case, an update operation checks at most $d(v) + d(u) + 1$ buckets since the gain difference of the vertices v and u is at most $d(v) + d(u)$. This is due to the fact that all edges have unit weights and then the gain of any vertex x is within the range $[g(x), -g(x)]$. Therefore, if the vertex v has gain $d(v)$ and the vertex u has the gain $-d(u)$ then the index m will be decremented exactly $d(v) + d(u)$ times. □

Now knowing the costs of all operations of a bucket priority queue, we can analyze the running time of the whole algorithm. The initialization of a bucket priority queue costs $\mathcal{O}(|V|)$ time. Now consider a move of a vertex v. After it was moved, the algorithm updates its $d(v)$ neighbors spending $\mathcal{O}(1)$ to calculate the new gain and $\mathcal{O}(1)$ to update its gain in the bucket priority queue. To extract the next candidate for the move, the algorithm spends $\mathcal{O}(d_{\text{avg}})$ amortized time. Since there are at most $\mathcal{O}(|V|)$ extraction operations, the total time of all extraction operations is $\mathcal{O}(|E|)$. Hence, the overall running of the algorithm is $\mathcal{O}(|V| + |E|)$. However, note that for weighted graphs this bound is $\mathcal{O}(|V| + |E|\omega_{\max}/\omega_{\min})$, where $\omega_{\max}$ and $\omega_{\min}$ are maximum and minimum weights in the graph, respectively.

Further Improvements and Extensions. A popular extension of the Fiduccia-Matheyses local search is its generalization to k-way graph partitioning using recursive bisection. Unfortunately, recursive bisection is not the best approach for k-way graph partitioning (see details in Section 3.4.3). Thus, there is a need for a direct k-way local search. For this, Sanchis [San93] as well as Hendrickson and Leland [HL95a] suggested to apply the Fiduccia-Matheyses local search to all pairs of blocks. The algorithm maintains $k(k-1)$ bucket priority queues (two for each pair of blocks) and performs moves in the following way. First all vertices with the highest gain in each priority queue are selected. Then vertices whose moves violate the balance constraint are discarded and the remaining vertex with the highest gain is selected. Then the move of the vertex is performed and the gains of its neighbors are updated. The algorithm, like the original Fiduccia-Matheyses local search, works in passes. The running time of one pass is $\mathcal{O}(k(|V| + |E|))$ since one needs $\mathcal{O}((k-1)|V| + |E|)$ time to calculate all the gains in the beginning of a pass and $\mathcal{O}(k|E|)$ time to update the gains of vertices during the pass. The authors also suggest to use only boundary vertices to initialize priority queues since only a small amount of vertices are moved during a pass in practice.

Cong and Kyu Lim [CL98] presented another generalization of the Fiduccia-Matheyses local search for k-way graph partitioning. The idea is to perform the Fiduccia-Matheyses local search only between pairs of blocks during each pass. This decreases the probability of local search to get stuck in a local minimum since the set of possible moves is reduced to the moves between pairs of blocks. The authors suggested several ways to select pairs of blocks to refine. Empirically, the best way is to pair blocks that had the maximum cut size reduction in the previous pass. The experiments indicate that this approach has better quality than the approach that uses $k(k-1)$ priority queues.

Karypis and Kumar [KK96] suggested several modifications of the Fiduccia-Matheyses local search that allow to reduce the running time of the algorithm. The first modification is a generalization of the Fiduccia-Matheyses local search for k-way graph partitioning. The idea is to maintain a global bucket priority queue that contains the best move to one of the blocks for each vertex. The second modification is a trade-off between running time and quality. The priority queue is initialized only with the vertices that have non-negative gains. Note that in the original algorithm the priority queue is initialized with all vertices. Another possibility suggested by Hendrickson and Leland [HL95a], Karypis and Kumar [KK95b] as well as Sanders and Schulz [SS11] is to initialize the priority queue with all boundary vertices. Then the algorithm works as before. During each pass, the algorithm extracts a vertex v with maximum gain in the global bucket priority queue and tries to move it to the corresponding block. If the move of v to that block does not violate the balance constraint then the algorithm performs the move and proceeds. Otherwise, the algorithm finds a block such that the gain of the move is maximized and the balance constraint is not violated. Next, it marks the vertex such that it cannot be moved anymore during current pass and inserts its non-marked neighbors to the priority queue that are not already in it. The algorithm stops when all vertices are moved or when x moves are performed without decreasing the cut size (the third modification). Then the algorithm rolls back to the best found partition and proceeds to the next pass. However, it is not clear how to choose the optimal parameter x since it may vary for different graphs. Note that the set of vertices moved by the aforementioned algorithm differs from that of the Fiduccia-Mattheyses local search which uses $k(k-1)$ priority queues. Furthermore, the Fiduccia-Mattheyses algorithm has better capabilities to climb out of local minima since it uses $k(k-1)$ priority queues and, thus, has more possibilities to move vertices.

The authors do not present a running time analysis for the algorithm. Depending on whether the algorithm updates the gains of the neighbors of the last moved vertex or not in the priority queue, the running time of one pass is $\mathcal{O}(|V| + |E| \log(\min(\Delta, k)))$ or $\mathcal{O}(|V| + |E|)$, respectively. A detailed analysis of the running time is presented in Section 4.2.3.

Another approach presented by Karypis and Kumar [KK96] is the greedy local search. The algorithm can be viewed as the label propagation algorithm (see Section 3.2.2)

that is used for refinement. The idea is to move only vertices with positive gains. Then there is no need in the priority queue since it is very likely that a vertex with a large positive gain will be moved even after the moves of the vertices with smaller gains. The algorithm works in passes and it iterates over boundary vertices in random order during a pass. Note that iterating in random order allows to construct different partitions by running the algorithm several times and then to choose the best partition. The move of a vertex occurs as follows. First, all moves that violate the balance constraint or have negative gains are discarded. Then the vertex is moved to the block with maximum gain. If several moves of the vertex to different blocks have maximum gain then the move that improves the balance the most is selected. The running time of one pass is $\mathcal{O}(|V| + |E|)$.

Osipov and Sanders [OS10] presented a localized modification of the k-way local search. Note that in this MGP algorithm $\Theta(|V|)$ contractions occur because edges are contracted one by one. The idea is that the local search algorithm starts around most recently uncontracted edge if its corresponding vertices are in the different blocks. Let there be a priority queue for each block that contains vertices and their corresponding gains to move them to this block. Then the local search algorithm works in iterations. During each iteration one vertex is moved as follows. The algorithm selects the vertex with maximum gain that can be moved without violating the balance constraint. After moving the vertex, the algorithm marks the vertex, inserts its unmarked neighbors to all priority queues if they were not inserted before or otherwise updates their gains. The algorithm proceeds until no improvement is possible. The running time of the algorithm is $\mathcal{O}(|V| + |E| \log(\min(\Delta, k)))$ since in the worst case we update gains of each neighbor in k priority queues. Additionally, Osipov and Sanders presented another stopping rule that allows to stop earlier than the stopping rule by Karypis and Kumar [KK96]. This is crucial for performance of their algorithm since $\mathcal{O}(|V|)$ local search are performed (one local search for each uncontracted edge). The adaptive stopping rule estimates the likelihood of a local search to find a better cut using an assumption that gains of all moves are identically distributed, independent random variables with the expectation μ and the variance σ^2 estimated in p previous moves. The authors note these assumptions are heuristic. Nevertheless, Osipov and Sanders show that given $p\mu > \alpha\sigma^2 + \beta$ then it is unlikely that the local search algorithm will find a better cut. Here α and β are tuning parameters. More precisely, β is initial number of moves during which stopping is not allowed. This prevents the stopping rule to stop if the variance happen to be small. The authors suggest to use $\beta := \ln |V|$. To show that reduction of a cut size is unlikely, the authors consider the maximum standard deviation from the expectation $(p + s)\mu$ after s steps $(p + s)\mu + \sqrt{s\sigma^2}$ and to stop when it is maximized with respect to s but still less than zero. This expression is maximized when $s = \frac{\sigma^2}{4\mu^2}$. Thus, $(p + s)\mu + \sqrt{s\sigma^2} < 0$ when $p > \frac{\sigma^2}{\mu^2}(\frac{x}{2} - \frac{1}{4})$ (note that $\mu < 0$) or simply $p\mu^2 \gg \sigma^2$.

Dutt and Deng [DD96] presented a localized modification of two-way local search. The idea is to insert all boundary vertices into a priority queue with gains as keys and

then set all keys to zero preserving the initial order of vertices. Then the algorithm extracts the top vertex v from the priority queue, moves it to the opposite block, and updates the keys of its neighbors in the priority queue in the following way. Assume that v is moved from a block A to a block B. If a neighbor is in the block B than the corresponding key is decremented by $w((v, u))$ since the edge (v, u) is not a cut edge anymore. Otherwise, the corresponding key is incremented by $w((v, u))$. Thus, the stronger connectivity of the neighbor in the block A to v the greater its corresponding key is. This allows to find dense subgraphs that are advantageous to move from A to B. The algorithm proceeds to extract vertices until the priority queue is empty and then reverts back to the best found partition. The running time of the algorithm is $\mathcal{O}(|V| + |E|)$.

Sanders and Schulz [SS11] presented a localized modification of k-way local search by Karypis and Kumar [KK96]. We refer to it as localized k-way multi-try local search. The idea is to initialize the priority queue with only one boundary vertex and to perform k-way local search around it. Localized k-way multi-try local search can find dense subgraphs whose moves decrease the cut size; whereas the original k-way local search cannot find such dense subgraphs since it extracts vertices scattered over the whole boundary and stops due to the stopping rule before moving these subgraphs. The running time of the modified local search remains the same as in the original k-way local search by Karypis and Kumar. Another advantage of localized k-way multi-try local search is that it can be parallelized, whereas the original k-way local search is known to be P-complete [SW91]. The idea is that each PE starts localized k-way multi-try local search around a random boundary vertex. Then it is unlikely that any two PEs will interact causing performance reduction. The parallelization of this algorithm is one of the main results of this thesis and was presented in [ASS18].

Meyerhenke et al. [MSS14] presented a label propagation algorithm with size constraints that can be used in the coarsening and uncoarsening phases. The detailed description of it is in Section 3.2.2. We only mention here that it can be used for refinement in the following way. Assume that we want to improve some partition of a graph. Then label propagation works as before but uses blocks of partitions as clusters to guarantee that the improved partition is balanced. Additionally, it is possible to use this algorithm for balancing in the following way. When we decide to which block to assign a vertex, we force to move it to the most underloaded block with the strongest connectivity; even if the connectivity to it is less than the connectivity to some other blocks.

3.5.3 Other Local Search Refinement Techniques

There are multiple other papers that present different modifications of local search. Hager et al. [HPD00] use quadratic programming to decide what subgraph is most advantageous to exchange between blocks. Ashcraft and Liu [AL97] presented a three steps approach to graph bipartitioning. First, the graph is partitioned into clusters (connected subgraphs) and so-called multisector, here a multisector is a set of vertices

after removal of which the clusters are disconnected from each other. Then a set of vertices whose removal leaves the graph separated into two connected components (bisector) is constructed from the multisector using the modified Kernighan-Lin local search that exchanges sets of vertices between blocks. Finally, the bisector is further improved using a graph matching algorithm.

Walshaw et al. [WCE95] presented a three-stage refinement technique. The first stage tries to improve the shape of the partition but it does not guarantee that the resulting partition will be balanced. To prevent the imbalance, the second stage applies a load balancing algorithm. Finally, the Fiduccia-Mattheyses local search is applied on the third stage to further improve the partition. Walshaw et al. [WC00a] presented a technique that first introduces a slight imbalance to find better partitions and then eliminates it using a load-balancing modification of the Fiduccia-Mattheyses local search. An interesting concept of helpful sets was presented by Monien et al. [DMP94; MD97; MS04] and described in the following section.

Helpful Sets. Monien et al. [DMP94; MD97; MPD00; MS04] presented another local search technique for the bipartitioning problem that is based on the idea of helpful sets. An l-helpful set is a set after moving of which to the opposite block the cut size decreases by l. First the idea of helpful sets were used in the proof of the upper bound on the size of a bipartition of a 4-regular graph by Hromkovic and Monien [HM91]. Monien and Diekmann [MD97] generalize the proof from [HM91] for $2k$-regular graphs of size n proving that the size of the minimum bipartition is at most $\frac{k-1}{2}n + 1$. They prove it by showing that if the current bipartition of a graph has size at least $\frac{k-1}{2}n + 2$ then there is a 4-helpful set in one block and (-2)-helpful set in the other block. Therefore, exchanging these sets between blocks decreases the cut size by two while preserving the balance. The authors employ the idea of helpful sets to refine bipartitions of general graphs. The algorithm works in passes as the Kernighan-Lin local search. During a pass, the algorithm searches for a l-helpful set in one block and $(-l + 1)$-helpful set in the other block of the same size. To find a l-helpful set, a BFS like algorithm is used. On each iteration this algorithm adds a vertex with the maximum helpfulness to the current helpful set. Here the helpfulness of a vertex is the decrease of the cut after moving it to the opposite block plus the sum of the edges to the vertices that are already in the current helpful set. Analogously, the helpfulness is defined for a set of vertices. To find a vertex with the maximum helpfulness, a bucket priority queue is used. After the vertex is added to the helpful set, the helpfulnesses of its neighbors is updated. The algorithm stops when at least l-helpful set is found. Afterwards, a $(-l + 1)$-helpful set of the same size is constructed in a similar way. If both sets are found they are exchanged and l is doubled; otherwise l is halved. Afterwards, the algorithm proceeds to the next pass. The algorithm stops when $l = 0$. The running time of one pass is $\mathcal{O}(|V| + |E|)$.

The experiments presented by Diekmann et al. [DMP94] show that the algorithm is comparable to the Kernighan-Lin local search in terms of quality and running

time. Monien and Schamberger [MS04] suggest several improvements that allow the algorithm to outperform the previous version of the algorithm and the Kernighan-Lin algorithm in terms of quality. Namely, they allow a slight imbalance and move only the helpful set with larger helpfulness.

Flow Based Refinement. Sanders and Schulz [SS11; SS12b] suggested a refinement technique that improves quality of partitions using maximum flows [FF56]. The idea is to select two adjacent blocks and find a new border between them reducing the cut size. Further, we discuss how to use the min cut problem to reduce the cut size while maintaining the balance constraint and how to select pairs of adjacent blocks.

Assume that the algorithm has selected two adjacent blocks V_1 and V_2. To perform the refinement, the algorithm uses a subgraph that belongs to both blocks, since this allows to reduce the running time and to maintain the balance constraint. In order to do this, two BFS are performed from the boundary vertices between two blocks. More precisely, the first BFS starts from the boundary vertices of V_1 and adds to the subgraph all the vertices and corresponding adjacent edges until it added $(1+\epsilon)|V|/k - c(V_2)$ vertices. Analogously, the second BFS starts from the boundary vertices of V_2 and completes the subgraph. Next, the algorithm finds a min cut of the subgraph using the max flow algorithm [CG97]. Finally, the boundary is adjusted with respect to the found min cut. Note that even if all vertices of the subgraph in V_1 move to V_2 then the balance constraint is still satisfied since $c(V_2) + (1+\epsilon)|V|/k - c(V_2) = (1+\epsilon)|V|/k$. The further details about how to choose the most balanced min cut are described in the original papers [HSS18; SS11; SS12b].

To select pairs of adjacent blocks, the algorithm uses the quotient graph as follows. In the beginning, all vertices of the quotient graph (blocks) are active. Next, the algorithm works in rounds. In the beginning of each round, it constructs a list of pairs of adjacent blocks, such that at least one block in each pair is active. Next, it marks all blocks as inactive and applies the aforementioned refinement technique for each pair. If any block changes then it is reactivated for the next round. The algorithm stops when all blocks are inactive.

3.5.4 Random Walks and Diffusion Processes

Meyerhenke et al. [MMS09b; MS12] suggested a refinement technique based on the bubble framework (see Section 3.4.2) that uses the ideas of random walks and diffusion processes, which distributes load values from source vertices to other vertices. The high-level idea is that it takes more time for a random walk to leave dense regions of a graph than sparse ones. The connection between random walks and diffusion processes explains the ability of the latter to distinguish dense and sparse regions. Namely, dense regions have load values greater than that of sparse regions. The authors use the diffusion process in the bubble framework to construct an initial partition, to refine an existing one and to find better source vertices. The modified bubble framework

works in iterations performing the following two steps during each of them. The first step works as follows. The algorithm starts by performing the diffusion process from a source vertex (initially selected at random) of each block. Next, it assigns the vertex to the block with maximum load value. After constructing (or improving) the partition, the algorithm proceeds with the second step independently selecting new source vertices for each block as follows. It performs the diffusion process using all vertices of a block as sources and selects the vertex with maximum load value as a new source. These steps are repeated until the algorithm converges. Note that this diffusion process can be used both for initial partitioning and refinement. In the following paragraph, we describe the details of the diffusion process.

A diffusion process distributes load values from the source vertices S (centers) to the remaining vertices of a graph G. During each iteration, the distribution process sends load from each vertex to its neighbors. In addition, each non-source vertex loses δ units of load and each source vertex receives $\delta|V|/|S| - \delta$ units of load. The new load values can be calculated using the following formula $w' = Mw + d$, where $M = I - \alpha L$ and $0 < \alpha \leq (\Delta + 1)^{-1}$. Here L is the Laplacian matrix of G where $L_{v,v} = \deg(v), L_{v,u} = -1 \ \forall (v,u) \in E$ and $L_{v,u} = 0$, otherwise. A load vector w contains load values of each vertex, and a drain vector d contains the loss of the load per iteration. In the beginning, load is evenly distributed over source vertices. The diffusion process stops when the load values stop to change. To find such the final distribution of load values, the linear system $Lw = d$ must be solved. Meyerhenke et al. [MMS09a; MMS09b; MS12] use algebraic multigrid (AMG) [Stü01; TOS00] to solve the aforementioned linear system.

Pellegrini [Pel07] suggested another diffusion process. The idea is to propagate k types of liquids from k sources corresponding to k blocks to other vertices. Note that some amount of each liquid annihilates the same amount of any other liquid. Moreover, each vertex loses some amount of liquid during each step. An established balance of liquids defines an improved partition of a graph. To decrease the running time and to find relevant sources, only a part of a graph is used. More precisely, the algorithm propagates liquids from k source vertices to vertices that are within a small distance from boundary vertices. In order to do this, the algorithm constructs the set of source vertices by merging the vertices of each blocks, which have shortest paths to the boundary vertices longer than three, into a super vertex and uses it as a source.

3.6 Parallel Refinement Techniques

In this section we consider different parallel refinement techniques that can be potentially used in an MGP algorithm or actually used in parallel graph partitioning frameworks.

3.6.1 Parallel Greedy Refinement

LaSalle and Karypis [LK13] presented a parallel shared-memory greedy local search algorithm which is a parallelization of the modified greedy local search from Section 3.5.2. The sequential algorithm works as follows. All boundary vertices are stored in a global priority queue with keys equal to their potential gains

$$\sum_{\substack{u \in N(v): \\ B[v] \neq B[u]}} w((v,u)) \quad - \sum_{\substack{u \in N(v): \\ B[v] = B[u]}} w((v,u)).$$

Afterwards, the algorithm extracts vertices from the priority queue and attempts to move them. To move a vertex, the algorithm computes $k-1$ gains; that is, for each $k-1$ possible moves to adjacent blocks. The move occurs if the maximum gain of the vertex is positive and the move does not violate the balance constraint. After moving the vertex, the algorithm updates the weights of the blocks and potential gains of its neighbors. The algorithm stops when the priority queue is empty. The running time of the algorithm is $\mathcal{O}(k|V| + |E|)$, since moving a vertex v requires $\mathcal{O}(k)$ time, extracting it from the priority queue $\mathcal{O}(1)$ time, and updating potential gains of its neighbors $\mathcal{O}(d(v))$ time. We now discuss the parallelization of the algorithm further.

Karypis and Kumar [KK99] presented a parallel distributed memory greedy local search algorithm which is a parallelization of the greedy local search from Section 3.5.2. The authors suggest to refine a partition using a coloring of the graph in the following way. Since all the vertices are randomly distributed over PEs, each PE processes its portion of vertices that belong to one color. When all PEs are finished, they proceed with the vertices of the next color and so on. Processing vertices that belong to the same color prevents the situations when the algorithm exchanges two connected vertices between blocks since this can increase the cut size. We do not describe the parallelization of this algorithm since we discuss its improved version in the next paragraph.

Karypis and Kumar [KK97] showed how to perform the aforementioned algorithm without coloring. Namely, they suggest to perform the local search algorithm in two phases. In the first phase, the vertices are allowed to move only from blocks with smaller IDs to the blocks with greater IDs. In the second phase the opposite moves are allowed. This distributed memory version of the algorithm is similar to the aforementioned shared-memory version except for a small detail. Namely, in the shared-memory version, each PE processes vertices in increasing order of their gains using a local priority queue to store them, whilst the distributed memory version processes vertices in arbitrary oder. Therefore, we describe only the shared-memory algorithm.

Shared-Memory Parallelization. To parallelize the sequential greedy local search algorithm, the authors suggest two approaches: *fine-grain refinement* and *coarse-grain*

refinement. Both approaches work in passes. During a pass, each PE has its own priority queue to store boundary vertices assigned to it and the pass ends when all priority queue are empty. The algorithm stops if the maximum limit of passes is exceeded or no vertex was moved in the previous pass.

The first *fine-grain refinement* approach maintains up-to-date block weights and connectivity of the vertices in the priority queue to the blocks. After extracting a vertex with the maximum potential gain, the algorithm checks if it is possible to move the vertex without violating the balance constraint and at least one move to some of its adjacent blocks has a positive gain. If both conditions are true then the algorithm locks the neighbors of the vertex and the blocks affected by the move. Then the algorithm checks that the move still does not violate the balance constraint and has a positive gain. If both conditions are true the move is performed, the connectivity information and block weights are updated and all locks are released. Although this approach is easy to implement, it has limitations: high synchronization overheads due to the locking, the number of blocks should be much greater than the number of PEs, and possibility of large number of false sharings due to the updates of the same connectivity information and block weights.

The second *coarse-grain refinement* works as follows. First, to prevent concurrent moves of connected vertices, each pass is split into phases such that the moves between two blocks are allowed only in one direction during a phase. Another difference from the previous approach is to use update buffers instead of locking. More precisely, a PE extracts a vertex v from the priority queue and decides to move it or not as in the first approach but without locking. Then it updates local neighbors (assigned to the same PE) of v. To update the neighbors of v that are assigned to other PEs, the PE stores the updates into the corresponding update buffers. After extracting a fixed number of vertices from its priority queue, all PEs communicate using update buffers to decide which moves to undo such that the balance constraint is not violated. After the decision is made, the remaining moves are confirmed and each PE updates connectivity information of assigned vertices. This process repeats until all priority queues are empty.

Additionally, the authors suggest three strategies to assign vertices to PEs: dynamic assignment, static assignment, and persistent assignment. The dynamic assignment tries to assign vertices maximizing load balance between PEs. The difference between static and persistent assignment is that static assignment assigns data to the PE each time a parallel task starts. This means that with the static assignment different sets of vertices can be assigned to one PE during different parallel task, whereas the persistent assignment guarantees that each vertex is always assigned to the same PE during the course of the whole algorithm. The experiments indicate that the persistent assignment shows better speed-ups than other assignments. The authors explain this by the better utilization of the data locality and caches. Furthermore, the authors discuss how load imbalance affects performance of the persistent assignment.

In summary, the parallel running time of one pass of the algorithm is $\mathcal{O}((|V|(\log|V| + k) + |E|)/p)$ given an even load balancing.

3.6.2 Parallel Hill-Climbing Refinement

Karypis and LaSalle [LK16] presented a parallel k-way local search algorithm called parallel hill-climbing refinement. The sequential version of the algorithm closely resembles localized k-way multi-try local search by Sanders and Schulz [SS11]. The sequential hill-climbing refinement algorithm tries to find a subgraph after moving of which to other block the cut size decreases. The algorithm works in passes. In the beginning of each pass, the algorithm inserts all boundary vertices into a *main* priority queue with their potential gains as keys. The potential gain is calculated using the following formula

$$gain(v) := \sum_{\substack{u \in N(v): \\ B[v] \neq B[u]}} w((v,u))/\sqrt{\Delta(v)} \quad - \sum_{\substack{u \in N(v): \\ B[v] = B[u]}} w((v,u)).$$

where $\Delta(v)$ is the number of external partitions connected to v. Thus, vertices connected to fewer blocks have larger potential gains than that of vertices connected to more blocks. Furthermore, the potential gain of v can be updated in $\mathcal{O}(1)$ time when a neighbor of v is moved. While the time per update is $\mathcal{O}(d(v))$ in the worst case if we use the following formula

$$gain(v) := \max_{b \in \mathcal{B}} \sum_{\substack{u \in N(v): \\ B[u] = b}} w((v,u)) \quad - \sum_{\substack{u \in N(v): \\ B[v] = B[u]}} w((v,u))$$

to calculate gains. The worst case is when a new block with maximum connectivity to v must be found. Afterwards, the algorithm performs iterations until the priority queue is empty. During each iteration, it extracts a vertex v from the *main* priority queue and tries to move it. If the vertex has a positive gain and the corresponding move does not violate the balance constraint then the algorithm moves the vertex. Otherwise, the algorithm tries to find a dense subgraph (a hill), that consists of v and vertices around it after moving of which the cut size reduces. The algorithm inserts v into an empty *auxiliary* priority queue with its gain as key. Next, the algorithm starts to repeatedly extract vertices from the *auxiliary* priority queue and adds them to the subgraph. Specifically, after extracting a vertex u from the priority queue, the algorithm adds u to the subgraph. If moving the subgraph to one of the blocks decreases the cut size then the algorithm moves it and proceeds to process the *main* priority queue. Otherwise, the neighbors of u are inserted to the priority queue or their gains are updated if they are already in the priority queue. Then the algorithm proceeds to extract vertices from the *auxiliary* priority queue and add them to the subgraph. The algorithm stops when the *auxiliary* priority queue is empty.

The authors suggested several optimizations to improve the running time of the algorithm. The first one is the stopping rule that signals to stop if $\sqrt{b(V)}$ subgraphs were empty. In this case, the algorithm was not able to find $\sqrt{b(V)}$ times a subgraph after moving of which the cut size reduces. Here $b(V)$ is the number of boundary vertices. The second optimization describes how to update the connectivity of a subgraph to blocks in constant time after moving a vertex from a block A and adding it to the subgraph. During the search of a subgraph, an array of size k is maintained that stores connectivity of the subgraph to each block. When the algorithm adds a new vertex v to the subgraph, it scans its neighbors updating the entries of the array in the following way. If a neighbor u of v is in the block $B \neq A$ then the algorithm adds $w((v,u))$ to the entry that corresponds to the block B. Otherwise, it subtracts $w((v,u))$ from the entry that corresponds to the block A. Then the reduction of the cut size moving the subgraph from the block A to some block B equals the difference of the corresponding entries.

The running time of one sequential pass is $\mathcal{O}(k|V| + |E| \log |V|)$, since moving a vertex requires $\mathcal{O}(k)$ time and searching for subgraphs requires $\mathcal{O}(|E| \log |V|)$ time in total.

Parallelization. To parallelize the sequential algorithm described above, the authors use similar techniques as in parallel greedy refinement (see Section 3.6.1). Each PE has its own *main* priority queue and *auxiliary* priority queue to find subgraphs. In the beginning of a pass, each PE inserts into its *main* priority queue the boundary vertices that are assigned to it. During a pass, each PE extracts a boundary vertex from its *main* priority queue and tries to find a subgraph around it after moving of which the cut size reduces. To prevent the possibility of two connected vertices to be moved by different PEs such that the edge remains in the cut, each pass is split into upstream and downstream phases and each block receives a random ID. During an upstream phase, only the moves from blocks with smaller ID to the blocks with greater IDs are allowed. During a downstream phase, the moves in opposite direction are allowed. When a PE finds and moves a subgraph, it can move vertices that are assigned to the other PE. Then the information about the current state of the partition is communicated to the owners of the moved vertices using message queues. When the *main* priority queues of all PEs are empty, the PEs synchronize and proceed to the next pass. During the search for subgraphs, PEs do not use any synchronization primitives, thus overlapping hills are possible. This means that the vertices that belong to both hills can be moved by both PEs. However, only one PE will move these vertices and this can potentially cause an increase of the cut size. The reason is that the remaining vertices of the separated hill may have a strong connectivity to the vertices from the overlapping region. The authors observe that such race conditions occur rarely and additional passes of the algorithm fix the problem.

Additionally, LaSalle et al. [LaS+15] presented a modified static assignment of vertices to PEs described in Section 3.6.1. They state that although such an assignment causes load imbalance, it allows to reuse data that is in caches. Furthermore, the authors

suggested a modification such that all boundary vertices of a block are assigned to one PEs. In order to do this, p buckets are created where each bucket contains boundary vertices assigned to corresponding PE. The buckets are constructed in the following way. Assume that in the beginning all vertices are assigned to PEs arbitrarily. Then each PE counts the number of vertices it must copy to each bucket. Next, a global prefix sum is computed such that each PE knows the starting position in each bucket at which the corresponding boundary vertices must be copied. After all boundary vertices are copied to buckets, each PE can access vertices assigned to it using its corresponding bucket. If $p > k$ the vertices of one block can be assigned to one of several PEs. To chose a particular PE to which a boundary vertex is assigned, the authors consider the external block to which the vertex is most connected.

The parallel running of the algorithm is $\mathcal{O}((k|V| + |E| \log |V|)/p)$ given an even load balancing, since moving a vertex requires $\mathcal{O}(k)$ time and searching for subgraphs requires $\mathcal{O}(|E| \log |V|/p)$ (PEs can move vertices that are not assigned to them during computations of subgraphs). Note that the authors assume that each PE processes approximately $\Theta(|V|/p)$ vertices and $\Theta(|E|/p)$ edges.

3.6.3 Parallel Label Propagation For Refinement

Meyerhenke et al. [MSS17] presented a parallel distributed memory label propagation algorithm that is used in the coarsening and uncoarsening phases. The detailed description of the algorithm is in Section 3.3.2. Here we describe only how the algorithm guarantees that after refinement each block does not exceed the threshold $(1 + \epsilon)\frac{|V|}{k \cdot f}$. In the beginning of the algorithm, the weights of blocks are calculated in the following way. Each PE calculates the weights of its local vertices within each block. Then all PEs communicate between each other the partial weights of blocks and sum them up. Eventually, each PE has the weights of all blocks and can perform the label propagation algorithm with size constraints. After all PEs finished one iteration of the label propagation algorithm, the PEs communicate to update weights of blocks. Note that it is still possible that several PEs moved vertices to some block and it is overloaded. The authors note that this approach is possible since k is much less then the number of the clusters during the coarsening phase and, thus, each PE can maintain and communicate an array of size k that contains the weights of blocks.

3.6.4 Other Parallel Distributed Memory Refinement Techniques

Parallelizations of Fiduccia-Mattheyses Local Search. A parallel distributed memory Fiduccia-Mattheyses local search algorithm was presented by Walshaw and Cross [WC00b]. The authors consider three different approaches to perform moves between blocks concurrently in the setting when each block is assigned to a particular PE. These approaches are different ways to tackle the problem of concurrent moves.

More precisely, how to prevent two processors to move connected vertices such that the cut size increases.

The first approach is called *interface optimization*. It performs moves between two connected blocks using one of the assigned PE. More precisely, consider a pair of blocks a and b and corresponding PEs PE_a and PE_b, respectively. PE_a performs moves if either $a < b$ and b is odd or $a > b$ and b is even. Assume without loss of generality that PE_A performs moves. In addition, let S_{ab} denotes the set of vertices whose moves from A to B have maximum gains. Analogously, the set S_{ba} is defined. Then PE_b sends the set S_{ba} to PE_a. Afterwards, PE_a performs the Fiduccia-Mattheyses local search starting from vertices $S_{ab} \cup S_{ba}$. When local search finishes, PE_a sends to PE_b a request for the vertices that moved from B to A. Additionally, it sends the vertices that moved from A to B. This approach has several disadvantages. The first one is that it requires additional communication from B to A when the local search algorithm moves a vertex from B to A and must know its neighbors in B. The second problem is that several PEs may perform moves from/to the same block and, thus, this approach does not necessarily construct a balanced partition.

The second approach, *alternating optimization*, is the same as the approach suggested by Karypis and Kumar [KK97]. The idea is to split a pass of the local search algorithm into two phases. During each phase, the moves between two blocks are allowed only in one direction. For example, during the first phase only the moves from a to b occur and during the second phase in the opposite direction. Specifically, the moves from a to b are only allowed if either $a < b$ and b is odd or $a > b$ and b is even. Otherwise, the moves from b to a are allowed. This method does not require additional communications since it does not need to know new boundary vertices from b. In addition, it does not need to request vertices from PE_b. In summary, the experiments indicate that this method constructs partitions of worse quality than the *interface optimization*.

Unlike the first two approaches, the third approach, *relative gain optimization*, allows to perform moves between pairs of blocks simultaneously. The authors suggest to prevent the problem of concurrent moves by using relative gain instead of conventional gain. Relative gain reflects the potential of a vertex to move to the opposite block. More precisely, if a vertex v has multiple neighbors in the opposite block with high gains of moves to the block of v then it is better to leave v in its block. Therefore, relative gain of a vertex is the difference between the gain of the vertex and the average gain of its neighbors in the opposite block. The parallel algorithm that uses this approach is simple. All PEs perform moves of vertices without taking into account moves of corresponding neighbors. After each PE marked vertices to be moved, all PEs communicate to exchange vertices. The authors observe that the moves of connected vertices occur rarely. In conclusion, the experiments indicate that this method constructs partitions of worse quality than the *interface optimization*.

Holtgrewe et al. [HSS10] presented another parallel distributed memory local search algorithm that performs the Fiduccia-Mattheyses local search between pairs of blocks.

The main idea of the algorithm is to select disjoint pairs of blocks and to refine them in parallel. In order to do this efficiently, the authors assume that the number of blocks is equal to p. Then there are two PEs for each pair of blocks and both PEs refine the blocks and then the best resulting partition is selected. There are two main problems: how to choose pairs of blocks to refine and how to perform refinement of two blocks in parallel given that each block resides on an individual PE.

The authors solve the first problem by considering an edge coloring of a quotient graph. The algorithm starts with constructing an edge coloring of the quotient graph using the parallel greedy algorithm [HSS10]. Afterwards, the algorithm iteratively considers each edge color and performs parallel refinement for pairs of blocks induced by the edge coloring.

The second problem is to refine blocks A and B. Recall that each block has an assigned PE that stores a subgraph belonging to this block. To perform the Fiduccia-Mattheyses local search, both PEs need to know subsets of vertices of opposite blocks. In order to do this, both PEs run a bounded BFS from the local boundary vertices between the blocks and exchange with each other the visited vertices. Then both PEs refine the blocks using the Fiduccia-Mattheyses local search and random seeds. After both local searches finish, the best partition is selected and PEs exchange vertices that have changed their blocks.

Parallelization of Diffusion Processes. Meyerhenke [MMS09a] presented a parallel shared-memory refinement technique based on the bubble framework and the diffusion process (see Sections 3.4.2 and 3.5.4). Since the bubble framework performs multiple runs of k independent diffusion processes for k blocks, they can be performed in parallel. But the scalability is limited by the number of blocks; that is, at most k PEs can be used simultaneously. Furthermore, Meyerhenke [Mey12] presented an improved distributed memory parallelization of the aforementioned refinement technique that overcomes the scalability issue by using all available PEs. More precisely, the linear systems within the diffusion process are solved using a conjugate gradient solver in combination with the traditional domain decomposition approach for parallelization.

Her and Pellegrini [HP10] presented a parallel distributed memory refinement technique based on the diffusion process (see Section 3.5.4) by Pellegrini [Pel07]. The idea of the sequential algorithm is to distribute liquid from k source vertices corresponding to k blocks to other vertices. In the parallelization, the number of source vertices changes. Specifically, each PE has k source super vertices. Source super vertices that correspond to the same block form a clique. More precisely, for every block each PE finds a portion of vertices assigned to it within a small distance from the boundary vertices of this block. Next, each PE connects them to its source super vertex that corresponds to this block. This source super vertex consists of the remaining vertices in this block assigned to this PE. Finally, the algorithm propagates liquids over the constructed graph improving the partition.

3.7 Multi-level Graph Partitioning Frameworks

Different variations of MGP scheme were independently developed and implemented by multiple researchers [Bou98; Gup97; HL95a; KK95b; KK98a; KK98c; MPD00; Pon+94; Pre01; Wal04; WC00a]. Furthermore, we briefly describe the details of these papers and corresponding implementations in chronological order. The detailed description of matching and clustering algorithms used in the frameworks are in Sections 3.2. The algorithms used in the initial partitioning phases of the frameworks are described in Section 3.4. Finally, the refinement techniques used in the uncoarsening phases of the frameworks are described in Section 3.5. Note that in some papers the old versions of the software packages are used as competitors since the latest versions were not released by that time.

Ponnusamy et al. [Pon+94] presented an MGP algorithm with the *heavy-edge matching algorithm* in the coarsening phase and either the simulated annealing algorithm [JS98] or the evolutionary/genetic algorithm [Kim+11] in the initial partitioning phase. They do not apply any refinement technique in the uncoarsening phase. Their experiments indicate that the contraction of a graph significantly decreases the time of its initial partitioning with a small loss in quality.

Hendrickson and Leland [HL95a] presented their MGP algorithm to solve the k-way graph partitioning problem. This algorithm consists of the *random matching algorithm* in the coarsening phase, the recursive spectral bisection [HL95b] in the initial partitioning phase, and the generalization of the *Fiduccia and Mattheyses local search* for k-way local search. The aforementioned algorithms were implemented by the authors in the `Chaco 2.0` framework. Their experiments indicate that this approach has better trade-off between running time and quality than the recursive interval bisection [Sim91; Wil91] with/without the Fiduccia and Mattheyses local search and the recursive spectral bisection with/without the Fiduccia and Mattheyses local search.

Gupta [Gup97] presented an MGP algorithm to solve the k-way graph partitioning problem. Gupta uses one of the following three matching algorithms in the coarsening phase: *heavy-edge matching*, *greedy matching*, and *heavy-triangle matching*. When the contracted graph in small enough, Gupta applies the graph growing algorithm to build the initial partition of it. Afterwards, either the *modification of Fiduccia and Mattheyses local search* or *greedy local search* is used to refine the partition in the uncoarsening phase. Additionally, Gupta repeatedly repartitions the contracted graph on each level of the graph hierarchy choosing the best partition to proceed with in the uncoarsening phase. The experiments indicate that for most of the instances the algorithm by Gupta is better than the frameworks `kMetis` and `pMetis` in terms of running time. But, on average, `kMetis` and `pMetis` construct cuts that are several percents smaller than the cuts constructed by the author's algorithm. Note that the comparison was performed with old versions of `kMetis` and `pMetis`.

Karypis and Kumar [KK95b; KK98a] presented an MGP algorithm for the graph bipartitioning problem. They consider four matching algorithms: *random matching*, *light-edge matching*, *heavy-edge matching*, and *heavy clique matching* in the coarsening phase. For the initial partitioning phase, they consider spectral bisection [BS93; HL95a; PSL90], the graph growing algorithm, and the greedy graph growing algorithm. For the uncoarsening phase, they use the *modified Fiduccia and Mattheyses local search*. The authors implemented this approach in the `pMetis` framework. According to their experiments [KK98a], the best trade-off between running time and quality shows the combination of the heavy-edge matching algorithm in the coarsening phase, the greedy graph growing algorithm in the initial partitioning phase, and the *modification of the Fiduccia and Mattheyses local search* in the uncoarsening phase. Additionally, they compare their best algorithm to the framework `Chaco 2.0` and the multi-level spectral bisection. The best algorithm outperforms both competitors in terms of running time and quality.

To solve the k-way graph partitioning problem, Karypis and Kumar [KK98c] suggest a modification of the MGP algorithm for the graph bipartitioning problem. Instead of partitioning a graph into k blocks using the multi-level recursive bisection, they first contract the graph and only then apply the multi-level recursive bisection in the initial partitioning phase. The authors implemented this approach in the `kMetis` framework. The experiments indicate that this approach outperforms the framework `Chaco 2.0`, the multi-level spectral bisection [KK98a], and their version of the multi-level recursive bisection in terms of running time and quality.

Walshaw et al. [WC00a] presented an MGP algorithm that uses *heavy-edge matching* in the coarsening phase to build a clustering and proceed to contract a graph until it has k vertices. Since the coarsest graph has k vertices, each vertex is assigned to its own block. In the uncoarsening phase, the authors allow a small amount of additional imbalance that decreases after each uncontraction of a graph. This allows to find high-quality partitions, although they can be imbalanced. To avoid highly imbalanced partitions, the amount of additional imbalance should be chosen carefully on each level of the contraction hierarchy. Otherwise, quality of a partition can significantly degrade during balancing of the partition. To perform balancing and local search simultaneously, the authors first compute a flow along the edges of the quotient graph, which determines how many vertices to transfer from one block to another to balance the partition. Next, *a modification of Fiduccia and Mattheyses local search* is employed. It tries to improve quality of the partition and in the same time to balance it using pre-computed flows. The aforementioned MGP algorithm is implemented in the `Jostle` framework. The experiments indicate that `Jostle` outperforms `kMetis` in terms of quality and running time on a benchmark of eight graphs.

Monien et al. [MPD00] presented an MGP algorithm to solve the graph bisection problem and implemented it in the `Party` framework. They consider four matching algorithms in the coarsening phase: *random edge matching*, *heavy-edge matching*, *greedy matching*, and *locally heaviest matching*. They contract a graph until it has

only two vertices, as a result there is no need for an initial partitioning of the graph. Afterwards, they apply the *helpful-sets refinement technique* in the uncoarsening phase. Their experiments indicate that the graph partitioning algorithm with the locally heaviest matching algorithm yields the best running time and quality. Additionally, they compare their algorithm to the following frameworks: `Chaco`, `Jostle`, and `pMetis`. But none of the software tools outperforms all the others in terms of quality and running time.

Pellegrini [Pel07] presents a diffusion-based MGP algorithm that computes partitions which have short length of boundary and do not have irregular shapes. In order to do this, the algorithm tries to improve the balance of a partition and refine it using the diffusion process. Afterwards, *Fiduccia and Mattheyses local search* is used to further improve the partition. The coarsening and initial partitioning phases are not described in details, however it is likely that the *heavy-edge matching algorithm* is used in the coarsening phase. This MGP algorithm is implemented in the `Scotch` framework. Pellegrini compares the algorithm to multi-level recursive bipartitioning algorithm from the `Scotch` framework and to the multi-level k-way MPG algorithm from the `kMetis` framework. The first set of experiments presents only a comparison of the algorithms from the `Scotch` framework on a benchmark of 11 graphs for different number of blocks. On average, the diffusion-based algorithm produces cuts with diameter and size less than those produced by the multi-level recursive bipartitioning algorithm. The second set of experiments presents a comparison of the algorithms from both frameworks for only two graphs and different number of blocks. Here the multi-level bipartitioning algorithm from the `Scotch` framework produces most of the smallest cuts but the diffusion-based algorithm produces more partitions with lower diameter. However, we consider this comparison as not comprehensive since it was made only for two graphs.

Meyerhenke et al. [MMS09a] presented an MGP algorithm based on the diffusion process and implemented it in the `DiBaP` framework. This algorithm first contracts a graph using the *local max algorithm* by Preis [Pre99]. When the contracted graph becomes sufficiently small the more expensive *algebraic multigrid (AMG) coarsening* [SSS12; Stü01; TOS00] is used to contract it further. During the initial partitioning phase, the authors use the *diffusion-based partitioner*. More specifically, it is the *bubble framework that employs a diffusion-based algorithm* to estimate how well two vertices are connected (the number of short paths between them). Afterwards, a similar algorithm is used to refine partitions of successive uncontracted graphs. When an uncontracted graph is sufficiently large, a modification of the aforementioned *diffusion based algorithm* is used for refinement. This modification reduces running time by concentrating only on the moves in the area of boundary vertices. The experiments indicate that the author's MGP algorithm partitions the eight largest graphs from Walshaw collection [Wal] with better quality than that of `kMetis` and `Jostle`.

Osipov and Sanders [OS10] presented an MGP algorithm with a novel n-level coarsening phase. This algorithm contracts only one edge per round in the coarsening phase. More

precisely, during each round an edge with the highest rank ($rank(u,v) = \frac{w(u,v)}{w(u)\cdot w(v)}$) is selected and contracted. The result is a coarse graph that has one vertex less than the finer one. When the number of vertices reduces by a preset factor $c > 2$, an additional trial with different seed is performed. Namely, the current coarse graph is partitioned two times with two different seeds and the best cut is taken. The experiments indicate that additional trials produce a small reduction of the cut sizes. When the coarsest graph has less than 20K vertices, the coarsening phase stops and the initial partition is constructed using the `Scotch` framework. In the uncoarsening phase, Osipov and Sanders perform *localized k-way local search* around the last uncontracted edge if at least one of its incident vertices is at the boundary. This local search may find an improvement moving only a small number of vertices since it is localized. But this also means that each of these local searches must perform a constant number of moves since there are $\mathcal{O}(|V|)$ coarse graphs and hence $\mathcal{O}(|V|)$ local searches are performed. To solve this problem, the authors developed an adaptive stopping rule. The experiments indicate that without the adaptive stopping rule running time increases by an order of magnitude giving only 1% quality improvement. The aforementioned MGP algorithm is implemented in the `KaSPar` framework. The authors compare `KaSPar` to `kMetis`, `Scotch`, and `KaPPa`. `KaSPar` produces considerably smaller cuts than other algorithms, although it is slower. Thus, additional tests are necessary that allow faster algorithms to compute more partitions and choose among them one with the smallest cut.

Sanders and Schulz [SS11] consider several algorithms based on the MGP scheme implemented in the `KaHIP` framework. The quality-oriented algorithm (**strong** configuration) produces cuts with the average size smaller than that of other competitors (`kMetis`, `DiBaP`, `Scotch`, and `KaSPar`). Additionally, the authors were able to improve partitions of multiple graphs from Walshaw collection [Wal]. This algorithm uses the *global path algorithm* to construct a matching with respect to the edge rating in the coarsening phase. The framework `Scotch 5.1.9` is used in the initial partitioning phase. In the uncoarsening phase, a combination of the *k-way and two-way Fiduccia and Mattheyses local searches*, the *two-way flow based refinement technique*, and the *localized k-way multi-try local search* is used.

Meyerhenke et al. [MSS14] considered multiple algorithms based on the MGP scheme and implemented them in `KaHIP`. Some of them are modifications of the MGP algorithms that were developed by the authors before in [SS11]. The quality-oriented algorithm (**strong** configuration) outperforms in terms of quality all competitors including previous MGP algorithms by the authors (`KaHIP`) and other competitors (`Scotch` and `kMetis`). The main novelty of these MGP algorithms is using the *size-constrained label propagation algorithm* to cluster a graph during the coarsening phase. Using a clustering of the graph in the coarsening phase allows to contract clusters of vertices and, as a result, quality of partitions improves. Multi-level recursive bisection is used in the initial partitioning phase, whereas in the uncoarsening phase, the same combination of the refinement techniques is used as before.

3.8 Parallel Multi-level Graph Partitioning Frameworks

In this section, we present a detailed overview of most existing parallel graph partitioning frameworks. The detailed description of matching and clustering algorithms used in the frameworks are in Sections 3.2. The algorithms used in the initial partitioning phases of the frameworks are described in Section 3.4. Finally, the refinement techniques used in the uncoarsening phases of the frameworks are described in Section 3.5.

`ParMetis` is a parallel distributed memory graph partitioning framework presented in multiple papers [KK96; KK97; KK98b; KK99]. In the coarsening phase, the authors use the parallel distributed memory *heavy-edge matching algorithm*. In the initial partitioning phase, to perform k-way partitioning the distributed coarse graph is collected on k PEs. Then the i-th PE performs $\mathcal{O}(\log k)$ multi-level recursive bisections; that is, until it constructs the i-th block [KK97]. In the uncoarsening phase, the authors use the parallel distributed memory *greedy refinement technique* to improve the partition. Furthermore, the authors also present an experimental evaluation of their parallel graph partitioning framework. The version of the parallel framework described by Karypis and Kumar [KK99] produces partitions of quality at most 5% worse than partitions produced by the parallel algorithm on a single PE. The authors also state that their parallel framework achieves speed-ups up to 35 for large graphs on 128 PEs.

`Jostle` is a parallel distributed memory graph partitioning framework presented in multiple papers [WC00b; WC+02; WC08; WCE97]. In the coarsening phase, the authors use the parallel distributed memory *heavy-edge matching algorithm*. When the distributed coarse graph is small enough, it is collected on each PE and partitioned using the sequential framework `Jostle`. Then the best partition is selected among all PEs [WC+02]. In the uncoarsening phase, the authors use the parallel distributed memory *Fiduccia-Mattheyses local search*. Additionally, Walshaw and Cross present an experimental comparison of the `Jostle` framework and the `ParMetis` framework [WC00b]. `Jostle` produces partitions that are on average 10% better but it is a factor of three slower than `ParMetis`.

`PT-Scotch` is a parallel distributed memory multi-level graph partitioning framework presented in multiple papers [CP08; HP10; Pel12]. In the coarsening phase, the authors use the parallel distributed memory matching algorithm [HP10]. When the distributed coarse graph is small enough, it is collected on each PE and additional contractions of the graphs are performed. Then each PE partitions its corresponding coarse graph using the `Scotch` framework [Pel12]. In the uncoarsening phase, the authors use the parallel distributed memory diffusion-based refinement technique. The authors present an experimental comparison of `PT-Scotch` and `ParMetis` in [HP10]. `PT-Scotch` produces partitions of better quality for most of the test cases except when the number of blocks is high. In these test cases, `ParMetis` produces partitions of

slightly better quality than PT-Scotch. The authors explain this by the use of the recursive bisection in the initial partitioning phase. Furthermore, both frameworks show comparable performance.

Meyerhenke et al. [MMS09a] presented a parallel shared-memory version of the DiBaP framework (see Section 3.7). The author parallelizes *algebraic multigrid (AMG)* coarsening and *diffusion-based algorithms* that are used during the initial partitioning and uncoarsening phases. The average speed-up on the eight largest graphs from Walshaw collection [Wal] is 1.55.

Holtgrewe et al. [HSS10] presented parallel distributed memory multi-level graph partitioning framework called KaPPa. In the coarsening phase, the authors use a parallel distributed memory matching algorithm described in Section 3.3.1. When the distributed coarse graph is small enough, it is collected on each PE and partitioned using the Scotch framework with a random seed. Finally, the best partition is selected among all partitions constructed by all PEs. In the uncoarsening phase, the authors use the parallel distributed memory *Fiduccia and Mattheyses local search*. The authors present an experimental comparison of KaPPa with ParMetis and sequential frameworks Scotch and kMetis. KaPPa produces partitions that are on average about 30% better than partitions produced by ParMetis but KaPPa is at least an order of magnitude slower than ParMetis. Nevertheless, the authors note that additional repetitions of ParMetis increase average quality only by 3%. Furthermore, KaPPa produces partitions that are on average about 16% and 8% better than partitions produced by Scotch and kMetis, respectively.

Mt-Metis is a parallel shared-memory graph partitioning framework presented in multiple papers [LaS+15; LK13; LK16]. Some parts of Mt-Metis are based on the similar ideas from the ParMetis framework. During the coarsening phase, the input graph is contracted using the parallel shared-memory *heavy-edge matching algorithm* matching. After the coarse graph is small enough, it can be partitioned using one of the following parallelizations of the initial partitioning [LK13]. The first parallelization employs the parallel nature of the recursive bisection. Each PE bisects the coarse graph into two blocks and then the best bisection is selected. Next, one half of the PEs recursively partition the first block and the other half of the PEs the second block. In the second approach, each PE performs multiple k-way partitionings and the best partition among all PEs is selected. In the uncoarsening phase, several refinement techniques can be used: the parallel shared-memory *greedy refinement technique* or the *hill-climbing refinement technique*. LaSalle et al. [LaS+15] present an experimental comparison of Mt-Metis, ParMetis and PT-Scotch. Mt-Metis is about 3 and 7.5 times faster than ParMetis and PT-Scotch, respectively, and still produces partitions of comparable quality.

Meyerhenke et al. [MSS17] presented the parallel distributed memory multi-level graph partitioning framework ParHIP. In the coarsening phase, the authors use the parallel *label propagation algorithm*. When the coarsest graph distributed over the PEs is small enough, it is collected on each PE and partitioned by each PE using the parallel

distributed memory evolutionary algorithm [SS12a] as follows. In the beginning, each PE constructs local partitions and performs combine/mutation operations on them to improve the partitions. Afterwards, the PEs exchange their populations using randomized rumor spreading by Doerr et al. [DF11]. Finally, the best partition is selected among all partitions constructed by all PEs. In the uncoarsening phase, the authors use again the parallel *label propagation algorithm* to refine the partition. Meyerhenke et al. present an experimental comparison of `ParHIP` and `ParMetis`. Additionally, they mention that they also tested `PT-Scotch` but it showed worse results than `ParMetis` in terms of quality and running time. `ParHIP` produces partitions of better quality that are on average about 19% better than partitions produced by `ParMetis`. Furthermore, `ParHIP` is about 38% better on average in terms of quality and more than two times faster on average for social networks, whereas `ParHIP` is only about 3% better on average in terms of quality and five time slower on average for mesh type networks.

Meyerhenke [Mey12] presented a parallel distributed memory multi-level graph partitioning framework `PDiBaP`. In the coarsening phase, the author uses parallel distributed memory *local max algorithm*. When the distributed coarse graph is small enough, the parallel *diffusion process* is used to construct an initial partitioning and to further improve it in the uncoarsening phase. Meyerhenke presented an experimental comparison of `PDiBaP` with `ParMetis` and `Jostle`. `PDiBaP` produces partitions of better quality than `ParMetis` and `Jostle` but it is about 30 times slower than both of the competitors.

Sui et al. [Sui+10] presented a parallel shared-memory multi-level graph partitioning framework that parallelizes the sequential framework `kMetis`. More precisely, the authors use the Java-based parallel framework `Galois` that employs so-called amorphous data-parallelism. This data-parallelism considers every vertex of a graph as an active element performing computations on it and its neighbors. Sui et al. presented experiments comparing their partitioner with `kMetis` and `ParMetis`. The authors' framework has comparable quality to that of `kMetis`. Furthermore, it has better quality than `ParMetis`. However, it is not clear why it shows better quality since both parallel frameworks employ similar algorithms. Moreover, the framework shows better speed-ups than `ParMetis` but it is slower. This is not surprising since it is implemented using Java.

Slota et al. [SMR14] presented a parallel shared-memory graph partitioning framework `PuLP` and parallel distributed-memory graph partitioning framework [Slo+17]. These frameworks are able to perform multi-objective and multi-constraint graph partitioning. Both frameworks use the *label propagation algorithm* to construct an initial partitioning. However, it can be highly imbalanced. Next, the frameworks improve the objectives using *Fiduccia-Mattheyses local search* and change the partition to satisfy multi-constraints. The authors compare both frameworks with `ParHIP` and `ParMetis`. Both competitors produce partitions of better quality for the single-constraint single-objective graph partitioning problem but `PuLP` shows better running times.

Kirmani and Raghavan [KR13] presented a parallel distributed-memory graph partitioning framework that solves a relaxed version of the graph partitioning problem where no strict balance constraint is enforced. The blocks only have to have approximately the same size and, thus, the results are incomparable with non-relaxed graph partitioners. Note that the problem is easier than fulfilling a strict balance constraint. Furthermore, their approach attempts to obtain information on the graph structure by computing an embedding into the coordinate space using multi-level graph drawing. Afterwards, partitions are computed using a geometric algorithm.

Uganer and Backstrom [UB13] use a parallel distributed-memory *label propagation algorithm* to partition large networks. The authors do not use a multi-level scheme and rely on a given or random partition which is improved by combining the unconstrained label propagation approach with linear programming. This approach does not produce high-quality partitions.

3.9 Hardness Results and Approximations

In this section, we discuss papers that prove the NP-hardness of the graph partitioning problem and suggest different approximation algorithms with a guaranteed approximation ratio to solve the problem. Unfortunately, all such approximation algorithms are complicated and hard to implement. Furthermore, they tend to have larger running times and worse solution quality compared to those of the aforementioned heuristics. Nevertheless, the approximation algorithms have worst-case guarantees unlike heuristics.

Garey et al. [GJS74] as well as Hyafil and Rivest [HR73] showed that the decision version of the graph partitioning problem for $k = 2$ is NP-complete. More specifically, Garey et al. presented a reduction of the problem to the *max cut problem* that asks to find a cut of at least a preset size. Feige and Krauthgamer [FK02] presented an approximation algorithm with $\mathcal{O}(\log^{1.5} |V|)$ approximation ratio for general graphs with non-negative weighted edges in the case when $\epsilon = 0$ and $k = 2$. Additionally, this algorithm has a better ratio of $\mathcal{O}(\log |V|)$ for planar graphs. Furthermore, they extend it for an arbitrary constant k (here k is *not part* of an input) preserving the approximating ratio. However, the running time is exponential in k: $\mathcal{O}(n^{\mathcal{O}(k)})$, where n is the size of the input. Andreev and Räcke [AR04] presented an approximation algorithm with $\mathcal{O}(\log^2 |V|)$ approximation ratio for general graphs with non-negative edge weights for the case when k is *part* of the input and $\epsilon > 0$. Furthermore, they proved that there is no approximation algorithm with a finite approximation ratio and polynomial running time if $\epsilon = 0$ and k is *part* of the input unless P = NP. A similar result was shown by Nguyen and Jones [BJ92]. Here finite means not necessary a constant but any finite function of n where n is the size of an input. For the case $\epsilon \geq 1$, Even et al. [Eve+97] developed an approximation algorithm with $\mathcal{O}(\log |V|)$ ratio. The other modification of the graph partitioning problem is the minimum k-cut

problem that asks to find a partition of a graph into exactly k blocks minimizing the sum of the weights of the cut edges without any size constraints. If k is *part* of the input then the problem is NP-hard. Otherwise, there is a polynomial algorithm that solves the problem in $\mathcal{O}(|V|^{\mathcal{O}(k^2)}T(|V|,|E|))$ time, where $T(|V|,|E|)$ is the time to find a minimum (s,t)-cut of a graph with $|V|$ vertices and $|E|$ edges. Note that the algorithm is exponential in k. Wagner et al. [WW93] showed the dependency between the lower bound on the size of a block and the running time when a graph must be partitioned into two blocks minimizing the number of edges between them. If the size of each block must be at least some constant then there is a polynomial algorithm that partitions the graph into such two blocks. If the size of each block is $\Omega(|V|^{\epsilon})$, where $\epsilon \in (0,1]$, then the problem is NP-hard. Furthermore, the complexity class of the problem remains unknown if the size of each block must be $\Omega(\log|V|)$.

Hardness of Parallelization. Savage and Wlodka [SW91] prove P-completeness for the Kernighan-Lin local search and the the Fiduccia-Mattheyses local search. Both proofs reduce the *circuit value problem* to the corresponding local search technique. P-completeness of both local searches evidences that it is *unlikely* that efficient parallel algorithms for them exist, although there is no proof of that. Here a parallel algorithm is efficient if it solves a problem in polylogarithmic running time using a polynomial number of processors. The set of such problems defines the class NC [Lei14].

Chapter 4

4 Parallel Shared-Memory Multi-level Graph Partitioning

Partitioning a graph into k blocks of similar size such that few edges are cut is a fundamental problem with many applications. This problem arises in almost all parallel distributed-memory problems where there is an implicit (or explicit) graph G that represents interconnections within the input data. The most well-known example is the application of graph partitioning to distribute work among nodes of a compute cluster. Another good example is the distribution of the Facebook social graph among multiple servers [Sha+16]. The fewer edges whose vertices reside on different servers the smaller the query time as well as the load of the network and the servers. In particular, when you process a graph in parallel on k processing elements (PEs), you often want to partition the graph into k blocks of about equal size. In this thesis, we focus on a version of the problem that constrains the maximum block size to $(1 + \epsilon)$ times the average block size and tries to minimize the total cut size, i.e., the number of edges that run between blocks.

There is a need for shared-memory parallel graph partitioning algorithms that efficiently utilize all cores of a machine. This is due to the well-known fact that CPU technology increasingly provides more cores with relatively low clock rates in the last years.Moreover, executing distributed-memory graph partitioners on shared-memory machines may yield disappointing speed-ups. Techniques geared towards shared-memory systems would help. Furthermore, shared-memory parallel graph partitioning algorithms can in turn be used as a component of a distributed graph partitioner, which distributes parts of a graph to nodes of a compute cluster and then employs a shared-memory parallel graph partitioning algorithm to partition the corresponding part of the graph on a node level.

References. This chapter is based on the conference paper [ASS18] published together with Peter Sanders and Christian Schulz. The text was mainly written by Yaroslav Akhremtsev with the editing by Peter Sanders and Christian Schulz. The design and analyses of the algorithms were made by all authors. The algorithms were implemented by Yaroslav Akhremtsev.

Contribution: We present a high-quality shared-memory parallel multi-level graph partitioning algorithm that parallelizes all three of the MGP phases – coarsening,

initial partitioning and refinement – using `C++17` multi-threading. Our approach uses the parallel label propagation algorithm that is able to shrink large complex networks fast during coarsening. Furthermore, we consider the parallel local max matching algorithm by Birn et al. [Bir+13] instead of the label propagation algorithm since we expect that for several classes of graphs (e.g., mesh graphs) the usage of matching algorithms improves quality of partitions. Our parallelization of localized k-way multi-try [SS11] is able to obtain high-quality solutions and guarantee balanced partitions by performing the majority of the work in mostly independent local searches on individual PEs. Using *cache-aware hash tables*, we limit memory consumption and expect improvement of cache locality. Summarizing, our approach scales comparatively better than other parallel partitioners and has considerably higher quality which does not degrade with increasing number of PEs.

The rest of this chapter is organized as follows. We discuss the related work in Section 4.1. In Section 4.2, we explain in details the multi-level graph partitioning approach and the algorithms that we parallelize. Section 4.3 presents our approach to the parallelization of the multi-level graph partitioning phases. More specifically, we present a parallelization of size-constrained label propagation [MSS14] as well as a parallelization of local max matching algorithm in Section 4.3.1. Further, we present a parallelization of localized k-way multi-try local search [SS11] in Section 4.3.3. Section 4.4 describes further optimizations. An extensive experimental evaluation is presented in Section 4.5.

4.1 Related Work

There has been a considerable amount of research on graph partitioning so that we refer the reader to Chapter 3 for more detailed overview of the material. Here, we focus on issues closely related to our main contributions in this chapter. Almost all general-purpose methods that are able to obtain good partitions for large real-world graphs are based on the multi-level graph partitioning approach (MGP). The basic idea can be traced back to multigrid solvers for solving systems of linear equations [Sou35] but more recent practical methods are based on mostly graph theoretic aspects, in particular edge contraction and local search. There are many ways to create graph hierarchies such as matching-based schemes [Die+00] or variations thereof [AK06] and techniques similar to algebraic multigrid (e.g., [SSS12]). See Section 3.1 for more details. Well-known software packages based on this approach include `Chaco`, `DibaP`, `Jostle`, `KaHIP`, `KasPar`, `Metis`, `Party`, and `Scotch`. A more detailed description of sequential and parallel frameworks can be found in Sections 3.7 and 3.8. In the following, we present a high-level overview of well-known parallel frameworks.

Probably the fastest available distributed memory parallel code is the parallel version of `Metis`, `ParMetis` [KK99]. See details in Section 3.8. This parallelization has problems maintaining the balance of the blocks since at any particular time, it is difficult to say

how many vertices are assigned to a particular block. Furthermore, `ParMetis` only uses very simple greedy local search algorithms that do not yield high-quality solutions. `Mt-Metis` by LaSalle and Karypis [LK13; LK16] is a shared-memory parallel version of `ParMetis` that uses a hill-climbing technique during refinement (see Section 3.6). This local search method is a simplification of localized k-way multi-try local search [SS11] in order to make it fast. The idea is to find a set of vertices (hill) whose move to another block is beneficial and then to move this set accordingly. However, it is possible that several PEs move the same vertex. To handle this, each vertex is assigned a processor element (PE), which can move it exclusively. Other PEs use a message queue to send a request to move this vertex.

Sui et al. [Sui+10] presented a Java-based shared-memory implementation of the framework `ParMetis`. They state slightly better speed-ups compared to `ParMetis`. Unfortunately, we were not able to receive the source code upon request. Therefore, we cannot compare ourselves to this framework. Furthermore, `Mt-Metis` shows significantly better speed-ups and produces smaller cuts than `ParMetis` [LK13; LK16]. Therefore, we expect that `Mt-Metis` is superior than this framework.

`PT-Scotch` [CP08; HP10; Pel12], the parallel version of `Scotch`, is based on recursive bipartitioning. This is more difficult to parallelize than direct k-partitioning since in the initial bipartition, there is less parallelism available. The unused processor power is used to perform several independent attempts in parallel. The involved communication effort is reduced by considering only vertices close to the boundary of the current partitioning (band-refinement). `KaPPa` [HSS10] is a parallel matching-based MGP algorithm which is restricted to the case where the number of blocks equals the number of processors used. `DiBaP` [Mey12; MMS09a] is a multi-level diffusion-based shared-memory framework that is targeted at small- to medium-scale parallelism with dozens of processors.

The label propagation clustering algorithm was initially proposed by Raghavan et al. [RAK07]. See Section 3.2.2 for more details. A single round of simple label propagation can be interpreted as the randomized agglomerative clustering approach proposed by Catalyurek and Aykanat [CA99a]. The distributed-memory label propagation algorithm by Uganer and Backstrom [UB13] has been used to partition social networks. The authors do not use a multi-level scheme and rely on a given or random partition which is improved by combining the unconstrained label propagation approach with linear programming. This approach does not yield high-quality partitions [MSS17]. Sections 3.2.2 and 3.3.2 contain overviews of different sequential and parallel clustering algorithms (including label propagation) that can be used in the coarsening phase. Note that the authors of all described parallel shared-memory label propagation algorithms rely on the internal scheduler of the parallel libraries they use. However, there is a need for a parallel label propagation algorithm that is able to achieve a good load balance for an arbitrary distribution of vertex degrees since the distribution can be highly non-uniform. Furthermore, we believe that work stealing also yields good load balance but it is more complex.

The local max matching algorithm was proposed by Preis [Pre99]. We use the parallel local max matching algorithm by Birn et al. [Bir+13] in our graph partitioning algorithm since it is easy to implement and has good scalability. Sections 3.2.1 and 3.3.1 contain overviews of different sequential and parallel matching algorithms that can possibly be used in the coarsening phase. We note that there is a parallel matching algorithm called Suitor by Manne and Halappanavar [MH14] that has better scalability and computes better matchings. However, usually graph partitioning frameworks with label propagation algorithms are faster and construct partitions of better quality than frameworks with matching algorithms according to Meyerhenke et al. [MSS14]. Therefore, we mainly focus on the parallelization of the label propagation algorithm in the coarsening phase.

Meyerhenke et al. [MSS17] propose the `ParHIP` framework to partition large complex networks on distributed memory parallel machines. The partition problem is addressed by parallelizing and adapting the label propagation technique for graph coarsening and refinement. The resulting system is more scalable and achieves higher quality than the state-of-the-art systems like `ParMetis` or `PT-Scotch` [MSS17].

Slota et al. [SMR14] presented a parallel shared-memory graph partitioning framework `PuLP`. This frameworks uses the *label propagation algorithm* to construct an initial partitioning. However, it can be highly imbalanced. Additionally, the framework improves quality using Fiduccia-Mattheyses local search.

Kirmani and Raghavan [KR13] presented a parallel distributed-memory graph partitioning framework that solves a relaxed graph partitioning problem without an enforced balance constraint.

4.2 Multi-level Graph Partitioning

The multi-level graph partitioning approach (MGP) is used in almost all existing graph partitioning frameworks due to its ability to compute high-quality partitions relatively fast. MGP algorithms recursively contract a graph preserving its basic structure and producing a hierarchy of graphs. Afterwards, it applies an initial partitioning algorithm to the coarser graph and iteratively undoes the contraction, applying local search or other combinatorial optimization techniques at each level of the hierarchy. Figure 4.1 shows a high-level outline of MGP. The overview of the frameworks that use MGP can be found in Sections 3.7 and 3.8. Section 3.1 gives a brief description of the papers that analyze different properties of MGP.

We now give an in-depth description of the three main phases of a multi-level graph partitioning scheme: coarsening, initial partitioning, and uncoarsening. In particular, we describe the sequential algorithms that we parallelize in the following sections. Our starting point here is the fast social configuration of KaHIP [MSS14; SS11]. For the development of the parallel algorithm, we add localized k-way multi-try local

search scheme that gives higher quality, and improve it to perform less work than the original sequential version. The original sequential implementations of these algorithms are contained in the KaHIP graph partitioning framework. Additionally, we employ random tie-breaking whenever possible. This diversifies the search and yields improved solutions by repeated tries.

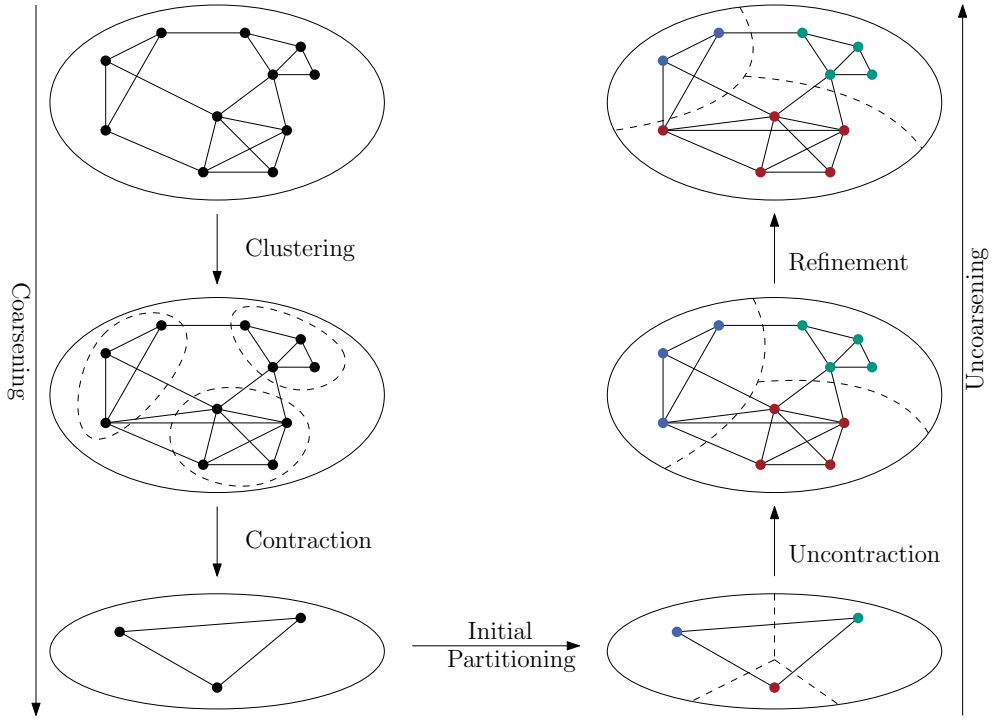


Figure 4.1: The multi-level graph partitioning scheme.

4.2.1 Coarsening

To create a new level of a graph hierarchy, the rationale here is to compute a clustering with clusters that are bounded in size and then to *contract* each cluster into a supervertex. If each cluster contains at most two vertices then the clustering is a matching. We can use either a matching or a clustering algorithm. This coarsening procedure is repeated recursively until the coarsest graph is small enough. Contracting a clustering (or a matching) works by replacing each cluster with a single vertex. The weight of this new vertex (or supervertex) is set to the sum of the weights of all vertices in the original cluster. There is an edge between two vertices u and v in the

contracted graph if the two corresponding clusters in the clustering are adjacent to each other in G, i.e., if the cluster of u and the cluster of v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weights of edges that run between cluster A and cluster B of the clustering. The graph hierarchy created in this recursive manner is then used by the partitioner. This construction ensures that a partition of the coarse graph corresponds to a partition of the finer graph with the same cut and balance. We now describe the clustering and the matching algorithms that we parallelize.

Clustering with Label Propagation

We denote the set of all clusters as C and the cluster ID of a vertex v as $C[v]$. There are a variety of clustering algorithms. See Section 3.2.2 for details. In our framework, we use the size-constrained label propagation algorithm by Meyerhenke et al. [MSS14].

The size constrained label propagation algorithm works in iterations, i.e., the algorithm is repeated ℓ times, where ℓ is a tuning parameter. Initially, each vertex is in its own cluster ($C[v] = v$) and all vertices are put into a queue Q in increasing order of their degrees. During each iteration, the algorithm iterates over all vertices in Q. A neighboring cluster $\mathcal{C}$ of a vertex v is called *eligible* if $\mathcal{C}$ will not become overloaded once v is moved to $\mathcal{C}$. When a vertex v is visited, it is *moved* to the eligible cluster that has the strongest connection to v, i.e., it is moved to the eligible cluster $\mathcal{C}$ that maximizes $\omega(\{(v, u) \mid u \in N(v) \cap \mathcal{C}\})$. If a vertex is moved to a different cluster then all its neighbors are added to a queue Q' for the next iteration. At the end of an iteration, Q and Q' are swapped, and the algorithm proceeds with the next iteration. It stops after a fixed number of iterations or when Q is empty. The sequential running time of one iteration of the algorithm is $\mathcal{O}(m + n)$.

Matching

A matching M is a set of edges that do not have common incident vertices. The weight of a matching is the sum of the weights of the edges it contains. We want to compute a matching with a weight that is as close as possible to the weight of the maximum weighted matching. Unfortunately, the algorithms that compute exact maximum weighted matching [Gab90] have super-linear running time, which we would like to avoid. Therefore, we use the local max matching algorithm by Preis et al. [Pre99]. Initially, the matching is empty and all vertices are put into a queue Q. During each iteration, the algorithm iterates over the vertices in Q. It selects for each vertex v an edge of maximum weight to a non-matched neighbor (local max neighbor) such that the total weight of both vertices does not exceed a preset threshold and breaking ties randomly. If the other vertex incident to the selected edge also selects this edge then these vertices are matched. Otherwise, the algorithm inserts v into the queue Q'. At the end of an iteration, Q and Q' are exchanged, and the algorithm proceeds with the

next iteration. It stops after a fixed number of iterations or when Q is empty. The sequential running time of one iteration of the algorithm is $\mathcal{O}(m + n)$.

4.2.2 Initial Partitioning

We adopt the algorithm from KaHIP [MSS14; SS11]. After coarsening, the coarsest graph in the hierarchy is partitioned into k blocks using a recursive bisection algorithm [Ker69]. More precisely, it is partitioned into two blocks and then the subgraphs induced by these two blocks are recursively partitioned into $\lceil \frac{k}{2} \rceil$ and $\lfloor \frac{k}{2} \rfloor$ blocks each. Subsequently, this partition is improved using local search and flow techniques. To get a better solution, the coarsest graph is partitioned into k blocks I times (a tunning parameter) and the best solution is returned.

4.2.3 Uncoarsening

After initial partitioning, a local search algorithm is applied to improve the cut of the partition. When local search has finished, the partition is projected to the next finer graph in the hierarchy, i.e., a vertex in the finer graph is assigned the block of its coarser representative. This process is then repeated for each level of the hierarchy.

There are a variety of local search algorithms: size-constrained label propagation, Fiduccia-Mattheyses k-way local search [FM82], localized k-way multi-try local search [SS11], max-flow min-cut based local search [SS11] etc. See section 3.5 for more details. The configurations of KaHIP use combinations of those. Since k-way local search is P-complete [SW91], our algorithm uses size-constrained label propagation in combination with localized k-way multi-try local search (LMLS). More precisely, the size-constrained label propagation algorithm can be used as a fast local search algorithm if one starts from a partition of the graph instead of a clustering and uses the size constraint of the partitioning problem. On the other hand, localized k-way multi-try local search is able to find higher quality solutions. Overall, this combination allows us to achieve a parallelization with good solution quality and parallelism.

We now describe LMLS. In contrast to previous k-way local search methods, LMLS is not initialized with *all* boundary vertices, that is, not all boundary vertices are eligible for movement at the beginning. Instead, the method is repeatedly initialized with a *single* boundary vertex. This enables more diversification and has a better chance of finding nontrivial improvements that begin with negative gain moves [SS11].

The algorithm is organized in a nested loop of global and local iterations. Each global iteration consists of multiple local iterations and works as follows. In the beginning, the algorithm constructs a hash table that contains all boundary vertices. We use a hash table since after each local iteration the set of boundary vertices changes and must be updated. Next, instead of putting *all* boundary vertices directly into a priority

queue, boundary vertices under consideration are put into a todo list T. Initially, all vertices are unmarked. Afterwards, the algorithm repeatedly chooses and removes a random vertex $v \in T$. If the vertex is unmarked, it performs k-way local search around v, marking every vertex that is moved during this search. More precisely, the algorithm inserts v and $N(v)$ into a priority queue using gain values as keys and marks them. Next, it extracts a vertex with a maximum key from the priority queue and performs the corresponding move updating the hash table with boundary vertices. Unmarked neighbors of the vertex are marked and inserted into the priority queue. If a neighbor of the vertex is already in the priority queue then its key (gain) is updated. Note that not every move can be performed due to the size constraint on the blocks. The local search around the vertex stops when the adaptive stopping rule by Osipov and Sanders [OS10] (see Section 3.5) signals to stop or when the priority queue is empty. In the end, the best partition that has been seen during the local search around the vertex is reconstructed. During local iteration, this is repeated until the todo list is empty. After a local iteration, the algorithm reinserts moved vertices into the todo list in random order. Next, if the *local* quantile-based stopping rule (see next paragraph) signals to stop or the cut size reduction during the last local iteration is zero then current global iteration finishes. Otherwise, the algorithm proceeds with the next local iteration. This approach allows further decreasing the cut size without a significant impact on running time. After the global iteration finishes, the algorithm uses the *global* quantile-based stopping rule to decide whether to start a new global iteration. The LMLS algorithm stops when the *global* quantile-based stopping rule signals to or the cut size reduction during the last global iteration is zero. This nested loop of local and global iterations is an improvement over the original LMLS search from [SS11] since they allow for a better control of the running time and quality of the algorithm.

The running time of one local iteration is $\mathcal{O}(|V| + \sum_{v \in V} d(v)^2)$ because each vertex can be moved only once during a local iteration and we update the gains of its neighbors using a bucket heap. The $d(v)^2$ term is the total cost to update the gain of a vertex v since we update the gain of a vertex at most $d(v)$ times. Note that this is an upper bound for the worst case and usually local search stops much earlier due the stopping rule or an empty priority queue.

Quantile-Based Stopping Rule. We developed a heuristic stopping rule that considers work-to-gain ratios of global (local) iterations to decide whether to perform a new global (local) iteration. Here work is the number of accesses to partition IDs of vertices during an iteration and gain is the reduction of the cut size performed during the iteration. We empirically observed for a subset of 14 graphs from our benchmark (see Section 2.3.2) that work-to-gain ratios have a distribution similar a log-normal distribution. Specifically, a random variable X is log-normally distributed if $\ln X$ is normally distributed. We denote the expectation of $\ln X$ as μ and the standard deviation as σ. Thus, the natural logarithm of a work-to-gain ratio has a distribution similar to a normal distribution. Now consider a sequence of work-to-gain ratios

$X_1, \ldots, X_n$ for n iterations and let $\hat{\mu} = \sum_{i=1}^{n} \ln X_i$ and $\hat{\sigma} = \sum_{i=1}^{n} (\ln X_i - \mu)^2 / \sqrt{n-1}$ be the estimates of μ and σ. To decide whether to stop or not after an $n+1$ iteration, we compute a quantile $Q(p)$ of the log-normal distribution using $\hat{\mu}$ and $\hat{\sigma}$ and stop if $Q(p) < X_{n+1}$ for some tuning parameter p. Here a quantile $Q(p)$ of the log-normal distribution is a value such that any random variable that has a log-normal distribution is less than or equal to $Q(p)$ with the probability p. Note that $Q(p)$ equals $e^{Q_N(p)}$ [Nor], where $Q_N(p)$ is a quantile of the normal distribution with the parameters $\hat{\mu}$ and $\hat{\sigma}$. Our heuristic assumption here is that it is likely that each subsequent improvement of the cut size requires more work than the previous one. Specifically, when we observe a sufficiently high work-to-gain ratio (greater than $Q(p)$) we expect further work-to-gain ratios to be even greater than the last one. Therefore, the amount of work we spent to reduce the cut size does not pay off and will only increase whereas the cut size reduction will decrease.

The *local* quantile-based stopping rule uses work-to-gain ratios computed during local iterations. Analogously, the *global* quantile-based stopping rule uses work-to-gain ratios computed during global iterations. Our experiments indicate that quantile-based stopping rule yield a fair trade-off between the running time and quality of the partition.

Figure 4.2 shows density histograms for natural logarithms of work-to-gain ratios of global iterations for four graphs from our benchmark set and sequences generated using normal distributions with expectations and standard deviations estimated using computed work-to-gain ratios. Specifically, if we compute a sequence of work-to-gain ratios $X_1, \ldots, X_n$, where n is the number of performed global or local iterations, then we plot a density histogram for a sequence $\ln(X_1), \ldots, \ln(X_n)$ and for a sequence generated using a normal distribution with the parameters $\mu = \sum_{i=1}^{n} \ln(X_i)/n$ and $\sigma = \sqrt{\sum_{i=1}^{n} (\ln(X_i) - \mu)^2/(n-1)}$. Furthermore, Tables 4.1, 4.2 show p-values generated using the Mann–Whitney U test (see Section 2.3.3) for 14 graphs. Our null hypothesis is that the sequence $\ln(X_1), \ldots, \ln(X_n)$ and the generated sequence both have the normal distribution. If the p-value is greater than 1% (significance level), then the null hypothesis is not rejected. We can see that most p-values are greater than 1% and, thus, we may assume that work-to-gain ratios have a distribution similar to a log-normal distribution. However, note that normality of $\ln(X_1), \ldots, \ln(X_n)$ is a purely empirical assumption since if the null hypothesis is not rejected it does not mean it is true. We can only assume that the difference between sequences is not practically significant according to the Mann-Whitney U test.

To strengthen our assumption even further, we consider Q-Q plots [Sci; Wik19] and compute 95% confidence intervals for the mean and the standard deviation of the difference between the approximated cdf function ($\hat{F}_n(x) = |\{\ln(X_i) \,:\, \ln(X_i) < x\}|/n$) and the cdf function ($F(x)$) of the normal distribution with parameters μ and σ. Specifically, we want to compute the mean and the standard deviation of the random variable $d(x) = |\hat{F}_n(x) - F(x)|$ and corresponding 95% confidence intervals. In order to do that, we compute $D_i = d(\ln(X_i))$ for every $i = 1 \ldots n$. Now $\mu_d = \sum_{i=1}^{n} D_i/n$

and $\sigma_d = \sqrt{\sum_{i=1}^{n}(D_i - \mu_d)^2/(n-1)}$. To compute confidence intervals for μ_d and σ_d, we use the basic bootstrap confidence limits [DH97, page 194, eq. (5.6)]. The idea behind this method is that if a sample of a distribution is large enough, then it has the same properties as the distribution. Therefore, we can sample the sample to compute confidence intervals. Specifically, we construct 1000 samples of size n from $\{D_1, \ldots, D_n\}$ using sampling with replacement and compute means (standard deviations) of all samples. Next, we sort computed means (standard deviations) and select $[q_{2.5}, q_{97.5}]$ as a corresponding confidence interval. Here $q_{2.5}$, $q_{97.5}$ are 2.5% and 97.5% quantiles of computed means (standard deviations). We can see that almost for all graphs the mean and the standard deviation of $d(x)$ is small (around 3.0% and 1.8% for global iterations and 3.1% and 2.2% for local iterations). Therefore, we conclude that the distribution of $\ln(X_1), \ldots, \ln(X_n)$ is similar to the normal distribution.

Q-Q plots 4.3 shows us quantiles[Sci] for four graphs and $k = 16, 64$. In each plot, the theoretical quantiles (quantiles of normal distribution) correspond to x-axis and data quantiles correspond to y-axis. We can see that for all graphs the points are around the line $y = x$. This means that quantiles of both distributions are approximately equal and, thus, the distribution of $\ln(X_1), \ldots, \ln(X_n)$ is similar to the normal distribution.

Table 4.1: Sample sizes ($|S|$), p-values, means (μ_d), standard deviations (σ_d) and corresponding confidence intervals of differences between approximated and real CDFs for global iterations. The last four columns are in percents.

Graph	k	$\|S\|$	p-value	μ_d	Conf. interval	σ_d	Conf. interval
ba_2_22	16.0	719	15.2	4.8	[4.5, 5.0]	3.1	[3.0, 3.2]
ba_2_22	64.0	1542	0.0	4.6	[4.5, 4.7]	2.5	[2.5, 2.6]
com-lj	16.0	378	25.3	2.5	[2.3, 2.7]	1.9	[1.8, 2.0]
com-lj	64.0	1595	14.7	3.0	[2.9, 3.1]	1.8	[1.7, 1.8]
com-orkut	16.0	856	6.4	3.2	[3.0, 3.3]	1.7	[1.7, 1.8]
com-orkut	64.0	1771	6.1	2.9	[2.8, 3.0]	1.9	[1.9, 1.9]
del_2_27	16.0	110	27.8	2.1	[1.9, 2.3]	1.2	[1.0, 1.3]
del_2_27	64.0	133	5.8	3.7	[3.3, 4.0]	2.1	[1.9, 2.3]
del_2_27_3d	16.0	237	29.4	5.6	[5.2, 6.0]	3.3	[3.1, 3.5]
del_2_27_3d	64.0	346	28.7	4.8	[4.5, 5.1]	3.0	[2.9, 3.2]
enwiki-2018	16.0	1561	17.7	2.3	[2.2, 2.3]	1.3	[1.2, 1.3]
enwiki-2018	64.0	1785	25.9	1.9	[1.8, 1.9]	0.9	[0.9, 0.9]
er-fact1.5-scale23	16.0	4426	1.5	2.6	[2.6, 2.7]	1.4	[1.4, 1.4]
er-fact1.5-scale23	64.0	1550	18.0	3.1	[3.1, 3.2]	1.9	[1.8, 1.9]
er_2_22_2_23	16.0	2857	0.3	4.8	[4.7, 4.9]	2.8	[2.8, 2.8]
er_2_22_2_23	64.0	1733	49.9	2.8	[2.7, 2.9]	1.8	[1.8, 1.8]
ljournal-2008	16.0	1257	41.6	3.2	[3.1, 3.3]	1.9	[1.8, 1.9]
ljournal-2008	64.0	1203	8.6	2.4	[2.3, 2.5]	1.3	[1.3, 1.3]
rgg_2_27	16.0	15	32.4	3.5	[2.0, 5.0]	2.9	[2.2, 3.9]
rgg_2_27	64.0	37	36.1	2.9	[2.4, 3.3]	1.5	[1.3, 1.8]
rgg_2_27_3d	16.0	105	12.7	3.0	[2.6, 3.4]	2.2	[2.0, 2.4]
rgg_2_27_3d	64.0	191	34.1	3.9	[3.6, 4.3]	2.5	[2.3, 2.6]
rhg_2_23	16.0	9	16.6	6.3	[4.1, 8.6]	3.7	[3.1, 5.2]
rhg_2_23	64.0	13	26.9	5.8	[3.6, 7.6]	3.9	[2.8, 5.8]
sk-2005	16.0	196	18.1	1.6	[1.5, 1.8]	1.1	[1.0, 1.2]
sk-2005	64.0	307	11.0	2.6	[2.5, 2.8]	1.6	[1.5, 1.7]
uk-2007	16.0	70	43.0	1.5	[1.2, 1.7]	1.0	[0.9, 1.2]
uk-2007	64.0	100	18.5	3.6	[3.2, 3.9]	1.8	[1.6, 2.0]
Harmonic mean				3.0	[2.7, 3.2]	1.8	[1.6, 1.9]

Table 4.2: Sample sizes ($|S|$), p-values, means (μ_d), standard deviations (σ_d) and corresponding confidence intervals of differences between approximated and real CDFs for local iterations. The last four columns are in percents.

Graph	k	$\|S\|$	p-value	μ_d	Conf. interval	σ_d	Conf. interval
ba_2_22	16.0	146	49.6	2.8	[2.5, 3.1]	1.7	[1.6, 1.9]
ba_2_22	64.0	384	2.5	3.9	[3.6, 4.2]	2.9	[2.8, 3.1]
com-lj	16.0	196	23.2	3.4	[3.1, 3.6]	2.0	[1.8, 2.1]
com-lj	64.0	148	50.0	5.8	[5.1, 6.4]	4.2	[4.0, 4.5]
com-orkut	16.0	182	40.2	2.5	[2.2, 2.7]	1.6	[1.4, 1.8]
com-orkut	64.0	169	38.5	2.5	[2.2, 2.8]	2.0	[1.8, 2.1]
del_2_27	16.0	12	39.8	5.4	[3.9, 7.0]	3.0	[2.2, 4.3]
del_2_27	64.0	14	36.5	3.3	[1.9, 4.8]	2.9	[2.5, 3.7]
del_2_27_3d	16.0	96	42.7	2.8	[2.4, 3.2]	2.1	[1.8, 2.4]
del_2_27_3d	64.0	133	10.1	4.2	[3.7, 4.6]	2.7	[2.5, 3.1]
enwiki-2018	16.0	130	2.1	2.5	[2.2, 2.9]	2.1	[1.9, 2.4]
enwiki-2018	64.0	214	47.1	1.7	[1.5, 1.9]	1.5	[1.4, 1.6]
er-fact1.5-scale23	16.0	575	5.7	7.6	[7.2, 7.9]	4.5	[4.3, 4.7]
er-fact1.5-scale23	64.0	242	39.4	2.4	[2.2, 2.7]	1.8	[1.7, 1.9]
er_2_22_2_23	16.0	82	20.7	2.2	[1.8, 2.5]	1.7	[1.4, 2.0]
er_2_22_2_23	64.0	3099	0.6	1.9	[1.8, 1.9]	1.2	[1.1, 1.2]
ljournal-2008	16.0	145	11.3	4.2	[3.6, 4.8]	3.8	[3.4, 4.3]
ljournal-2008	64.0	147	20.3	2.0	[1.7, 2.2]	1.6	[1.4, 1.9]
rgg_2_27	16.0	11	30.0	5.5	[3.3, 7.5]	3.8	[3.0, 5.0]
rgg_2_27	64.0	11	50.0	4.7	[3.4, 6.0]	2.4	[1.9, 3.4]
rgg_2_27_3d	16.0	29	32.6	3.5	[2.7, 4.2]	2.4	[1.8, 3.0]
rgg_2_27_3d	64.0	34	41.0	2.9	[2.2, 3.5]	1.9	[1.5, 2.4]
rhg_2_23	16.0	8	47.9	8.2	[2.9, 12.4]	7.5	[5.2, 12.7]
rhg_2_23	64.0	5	41.7	12.1	[3.2, 19.0]	10.0	[7.3, 18.3]
sk-2005	16.0	59	47.4	1.7	[1.4, 2.0]	1.3	[1.1, 1.6]
sk-2005	64.0	64	26.2	3.2	[2.7, 3.7]	1.8	[1.5, 2.2]
uk-2007	16.0	34	39.6	3.1	[2.4, 3.9]	2.2	[1.9, 2.7]
uk-2007	64.0	36	38.7	4.1	[2.9, 5.2]	3.3	[2.7, 4.1]
Harmonic mean				3.1	[2.5, 3.6]	2.2	[1.9, 2.6]

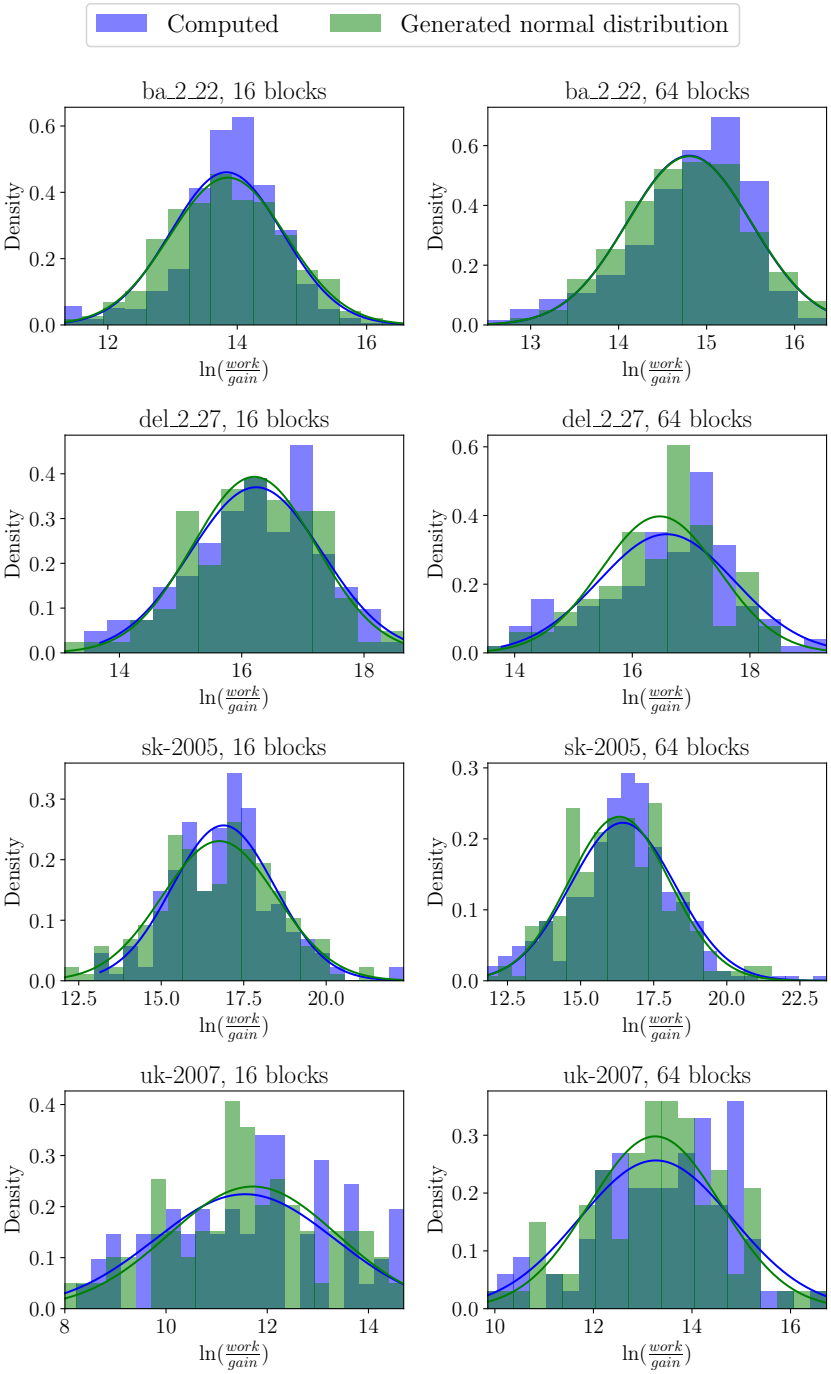


Figure 4.2: Density histograms for different types of graphs.

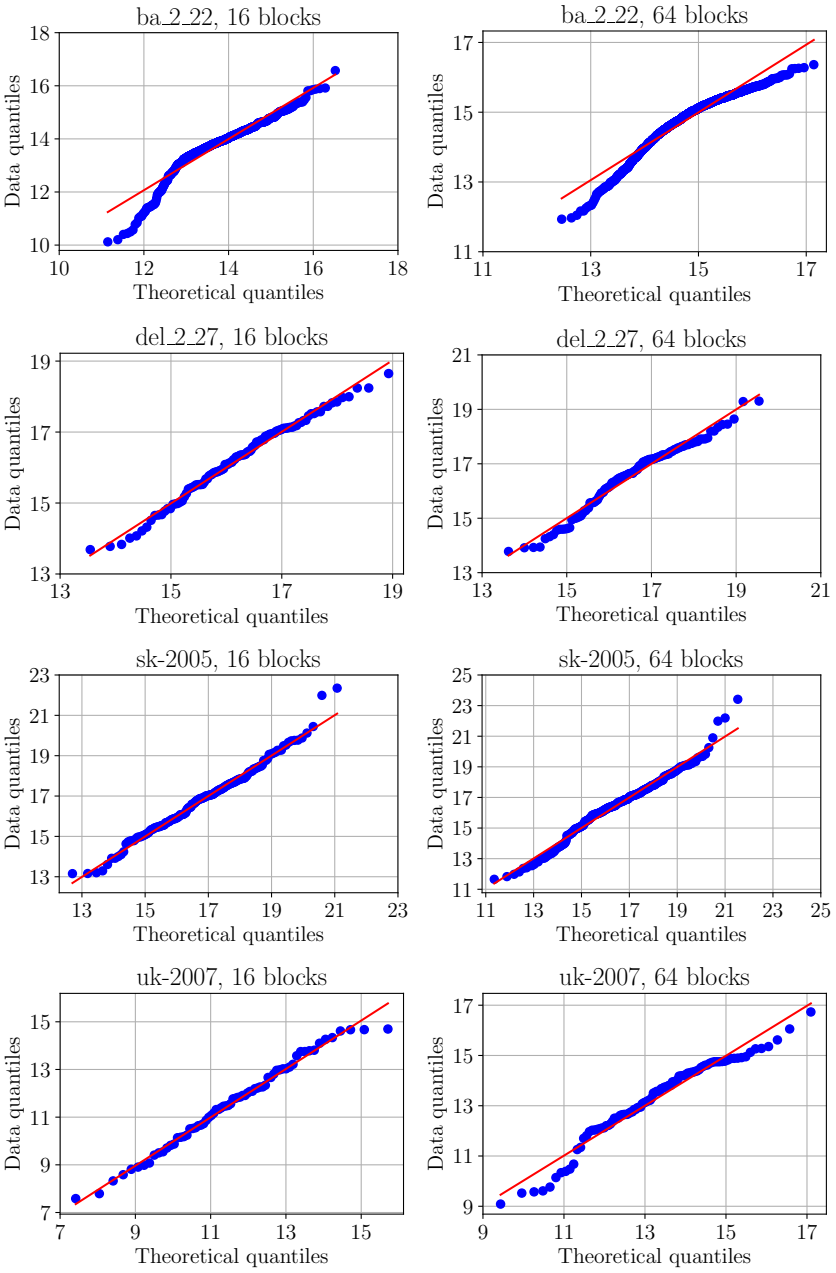


Figure 4.3: Q-Q plots for different types of graphs.

4.3 Parallel Multi-level Graph Partitioning

Running time analysis of the sequential algorithm shows that each of the components of the MGP scheme has a significant contribution to the overall execution time. See Section 4.5.5 for more details. Hence, to achieve good (parallel) speed-ups, we have to parallelize all phases. In this section, we describe our parallelization. The section is organized along the phases of the MGP scheme: first we show how to parallelize coarsening, then initial partitioning and finally uncoarsening. Our general approach is to avoid bottlenecks as well as performing independent work as much as possible.

4.3.1 Coarsening

In this section, we present the parallel version of the size-constrained label propagation algorithm to build a clustering, the parallel local max matching algorithm, and the parallel contraction algorithm.

Parallel Size-Constrained Label Propagation

To parallelize the size-constrained label propagation algorithm, we adapt a clustering technique by Staudt and Meyerhenke [SM16] to coarsening. Their algorithm iterates in parallel over a set of active vertices; i.e., vertices whose neighbors changed their labels on previous iteration, and assigns them to new clusters. Initially, we sort the vertices by increasing degree using the fast parallel sorting algorithm by Axtmann et al. [Axt+17]. We then form work packets representing a roughly equal amount of work and insert them into a TBB (threading building blocks) concurrent queue Q [Tbb]. Note that we also tried the work-stealing approach [SSP07] but it showed worse running times. Our constraint is that a packet contains vertices with a total number of neighbors at most B. We set $B = \max(1000, \sqrt{|E|})$ in our experiments – the 1 000 limits contention for small graphs and the term $\sqrt{|E|}$ further reduces contention for large graphs. Additionally, we have an empty queue Q' that stores packets of vertices for the next iteration. During an iteration, each PE checks if the queue Q is not empty, and if so it extracts a packet of vertices from the queue. A PE then chooses a new cluster for each vertex in the currently processed packet, breaking ties randomly. A vertex is then moved if the cluster size is still feasible to take on the weight of the vertex. Cluster sizes are updated atomically using a CAS instruction. This is important to guarantee that the size constraint is not violated. Neighbors of moved vertices are inserted into a packet for the next iteration. If the sum of vertex degrees in that packet exceeds the work bound B then this packet is inserted into queue Q' and a new packet is created for subsequent vertices. When the queue Q is empty, the main PE exchanges Q and Q' and we proceed with the next iteration. Pseudocode of the parallel algorithm can be found in Algorithm 4.1.

One iteration of the algorithm can be done with $\mathcal{O}(|V| + |E| + p^2 + p\Delta)$ work and in $\mathcal{O}((|V| + |E|)/p + p + \Delta)$ parallel time.

Parallel Matching

We adapt the parallel local max matching by Birn et al. [Bir+13] to the shared-memory model. The algorithm works in iterations and each iterations consists of three parallel phases. The algorithm maintains a TBB concurrent queue Q [Tbb] with blocks of not matched vertices as in the parallel size-constrained label propagation. During the first phase, each PE extracts vertices from Q, finds their local max neighbors and inserts them to an auxiliary concurrent queue Q'. In the second phase, each PE extracts vertices from Q' and tries to match them with their local max neighbors. A matching of a vertex v with its local max neighbor u occurs if the local max neighbor of u is v. If the matching does not occur then the PE inserts v to Q and proceeds. In the third phase, we remove edges between unmatched vertices in Q and matched vertices. Specifically, we remove all edges between v and matched vertices in $\mathcal{O}(d(v))$ time by moving the IDs of the matched vertices to the end of the adjacency list of v. Afterwards, the algorithm proceeds with the next iteration. This continues until no vertex was matched in an entire iteration. Pseudocode of the parallel algorithm can be found in Algorithm 4.2. One iteration of the algorithm can be done with $\mathcal{O}(|V| + |E|)$ work and in $\mathcal{O}((|V| + |E|)/p + \log p + \Delta)$ parallel time. Furthermore, on average the algorithm removes at least half of the edges when processing graphs with unit edges weights or with random uniform edge weights. Thus, in this case the total work and parallel time over all iterations are $\mathcal{O}(|V| + |E|)$ and $\mathcal{O}((|V| + |E|)/p + \log p + \Delta)$, respectively.

Parallel Contraction

The contraction algorithm takes a graph $G = (V, E)$ as well as a clustering C and constructs a coarser graph $G' = (V', E')$. The contraction process consists of three phases: the remapping of cluster IDs to a consecutive set of IDs, edge weight accumulation, and the construction of the coarser graph. The remapping of cluster IDs assigns new IDs in the range $[0, |V'| - 1]$ to the clusters, where $|V'|$ is the number of clusters in the given clustering. We do this by calculating a prefix sum on an array that contains ones in the positions equal to the current cluster IDs. This phase runs in $\mathcal{O}(|V|)$ time when it is done sequentially. Sequentially, the edge weight accumulation step calculates weights of edges in E' using a hash table. More precisely, for each cut edge $(v, u) \in E$ we insert a pair $(C[v], C[u])$ into the hash table and accumulate weights for the pair if it is already contained in the table. Due to insertion of the cut edges into the hash table, the expected running time of this phase is $\mathcal{O}(|E|)$. To construct the coarse graph we iterate over all edges E' contained in the hash table.

Algorithmus 4.1: Parallel Size-Constrained Label Propagation

Input: graph $G = (V, E)$; cluster-size upper bound U; iterations ℓ
Output: clustering C

```
1  V_s = sort(V)                          // parallel sort vertices by degree in increasing order
2  maxBlockSize = max(√|E|, 1000)         // maximum block size
3  Q = V_s                                // concurrent queue
4  Q' = ∅                                 // queue for the next iteration
5  C = {0, ..., |V| − 1}                  // init clustering
6  S = {1, ..., 1}                        // sizes of clusters
7  inQ = {False, ..., False}              // array of bits to mark vertices in Q
8  inQ' = {False, ..., False}             // array of bits to mark vertices in Q'
9  for i = 1 to ℓ
10   while Q is not empty do in parallel
11     Block B = Q.pop()
12     Block B' = ∅                       // new block of active vertices
13     foreach v ∈ B do
14       inQ[v] = False
15       HashMap map = ∅                  // connection strengths
16       foreach u ∈ N(v) do  map[C[u]] += w(v, u)
         // select strongest connection
17       bestCluster = C[v]
18       bestSize = S[bestCluster]
19       foreach cluster ∈ map do
20         s = S[cluster]
21         if map[cluster] ≥ map[bestCluster] and s + w(v) < U then
22           bestCluster = cluster; bestSize = s
         // move v to bestCluster and update size
23       if C[v] ≠ bestCluster then
24         move = True
25         do
26           bestSize = S[c]
27           if bestSize + w(v) > U then go to line 13
28         while not CAS(S[c], bestSize, bestSize + w(v))
29         S[C[v]] −= w(v)               // atomically
30         C[v] = c
31         foreach u ∈ N(v) do
32           if not inQ[u] then          // atomically
33             inQ[u] = True             // atomically
34             B' = B' ∪ {u}
35             if Σ_{u∈B'} deg(u) > maxBlockSize then
36               Q'.push(B'); B' = ∅
37     if Q is empty and |B'| ≠ 0 then Q'.push(B')
38   exchange inQ and inQ'
39   exchange Q and Q'
```

Algorithmus 4.2: Parallel Local Max Matching

Input: graph $G = (V, E)$; matching-size upper bound U
Output: matching M

```
1  maxBlockSize = max(√|E|, 1000)                          // maximum block size
2  Q = V                                                   // concurrent queue
3  Q' = ∅                                                  // queue for the next iteration
4  M = {0, ..., |V| − 1}                                   // init matching
5  S = {notMatched, ..., notMatched}                       // statuses of vertices, |S| = |V|
6  while Q ≠ ∅ do
7      while Q is not empty do in parallel
8          Block B = Q.pop()
9          foreach v ∈ B do
10             z = −1
11             z = argmax_{u∈N(v)}{w(v,u) : S[u] = notMatched and w(u) + w(v) < U}
12             if z ≠ −1 then M[v] = z
13         Q'.push(B)
14     while Q' is not empty do in parallel
15         Block B = Q'.pop()
16         Block B' = ∅                                    // new block of vertices
17         foreach v ∈ B do
18             match = M[v]
19             if M[match] = v then S[v] = S[match] = matched
20             else
21                 B' = B' ∪ {v}
22                 if Σ_{u∈B'} deg(u) > maxBlockSize then
23                     Q.push(B'); B' = ∅
24         if Q' is empty and |B'| ≠ 0 then Q.push(B')
25     remove edges between unmatched vertices in Q and matched vertices
26     if no vertex was matched during the last iteration then break
```

This takes time $\mathcal{O}(|V'| + |E'|)$. Hence, the total expected running time to compute the coarse graph is $\mathcal{O}(|V| + |E|)$ when run sequentially.

The parallel contraction algorithm works as follows. In the first phase, we remap the cluster IDs using the implementation of the parallel prefix sum algorithm by Singler et al. [SSP07]. In the second phase, edge weights are accumulated by iterating over the edges of the original graph in parallel. We use the concurrent hash table of Maier et al. [MSD19], initializing it with a capacity of $\min(2|E|/|V| \cdot |V'|, |E|/10)$. Note that this is a rough estimation of $|E'|$ and in case it underestimates the real value the concurrent hash table is able to grow. We use the average degree $2|E|/|V|$ of G

since we expect that the average degree of G' does not increase. In the third phase, we construct an array of coarse edges and an array of offsets into the array of coarse edges. To parallelize the third phase, we first calculate degrees of coarse vertices by iterating over the concurrent hash table in parallel. Then we use the parallel prefix sum algorithm to compute offsets of all coarse vertices into the array of coarse edges. Finally, we construct the array of coarse edges by iterating over the concurrent hash table in parallel one more time. The parallel contraction algorithm runs in expected $\mathcal{O}((|V| + |E|)/p + \log p)$ parallel time. Pseudocode of the algorithm can be found in Algorithm 4.3.

Algorithmus 4.3: Parallel Contraction

Input: graph $G = (V, E)$; clustering C
Output: contracted graph $G' = (V', E')$

```
// First phase:
1  bitMark = {0, ..., 0}                    // Array of size |V| with a bit for each cluster ID
2  for v ∈ 0, ..., |V| − 1 do in parallel  bitMark[C[v]] = 1
3  ∀ v ∈ [1, |V|) :  C[v] = Σ_{u=0}^{C[v]−1} bitMark[u]    // using parallel prefix sum algorithm
   // Second phase:
4  HashMap H = ∅
5  for v ∈ V do in parallel
6      for u ∈ N(v)
7          if C[v] ≠ C[u] then H[(C[v], C[u])]  += w(v, u)

   // Third phase:
8  D = {0, ..., 0}                          // The degrees of vertices V', |D| = |V'|
9  for (v, u) ∈ H do in parallel  D[v]++
   // V' is an array of offsets into E'. The adjacency list of v starts from E'[V'[v]]
10 ∀ v ∈ [1, |V'|) : V'[v] = Σ_{u=0}^{v−1} D[u]; V'[0] = 0      // using parallel prefix sum
     algorithm
11 E' = {(0, 0), ..., (0, 0)}                                  // array of edges E'
12 for (v, u, weight) ∈ H do in parallel
       // atomically adds 1 and returns previous value
13     e = fetch_add(V'[v], 1)
14     E'[e].target = u
15     E'[e].weight = weight
```

4.3.2 Initial Partitioning

To improve the quality of the resulting partitioning of the coarsest graph $G' = (V', E')$, we partition it into k blocks $\max(p, I)$ times instead of I times. We perform each

partitioning step independently in parallel using different random seeds. To do so, each PE creates a copy of the coarsest graph and runs KaHIP sequentially on it. Assume that one partitioning can be done in time T. Then $\max(p, I)$ partitions can be built with $\mathcal{O}(\max(p, I) \cdot T + p \cdot (|E'| + |V'|))$ work and in $\mathcal{O}(\frac{\max(p,I) \cdot T}{p} + |E'| + |V'|)$ parallel time, where the additional terms $|V'|$ and $|E'|$ account for the time each PE copies the coarsest graph.

4.3.3 Uncoarsening/Local Search

Our parallel algorithm first uses size-constrained parallel label propagation to improve the current partition and afterwards applies our parallel localized multi-try k-way local search (LMLS). The rationale behind this combination is that label propagation is fast and easy to parallelize and will find and apply all the easy improvements. Subsequently, LMLS will then invest considerable work to find a few nontrivial improvements. In this combination, only few vertices actually need be moved globally which makes it easier to parallelize LMLS scalably. When using the label propagation algorithm to improve a partition, we set the upper bound U to $L_{\max}$, the size constraint of the partitioning problem.

Parallel LMLS works in a nested loop of local and global iterations as in the sequential version. Recall that each global iteration consists of multiple local iterations. Initialization of a global iteration copies all boundary vertices to a consumer/producer queue Q that consists of local buckets that correspond to PEs. Since a set of boundary vertices is implemented using a hash table, this guarantees that vertices copied to buckets are in random order. The consumer/producer queue Q implements the todo list T used in the description of the sequential LMLS in Section 4.2.3. During a local iteration, each PE extracts vertices from the consumer/producer queue Q. Afterwards, it performs *local* moves around them; that is, global block IDs and the sizes of the blocks remain *unchanged*. Specifically, each PE locally moves the vertex extracted from Q and vertices around it. When the consumer/producer queue Q is empty, the algorithm applies the best found sequences of moves to the global data structures and reinserts into Q moved vertices. LMLS decides whether to start a new local iteration, a new global iteration or to stop in the same manner as its sequential version. Pseudocode of one global iteration of the algorithm can be found in Algorithm 4.4. In the paragraphs that follow, we describe how to construct and maintain the set of boundary vertices, our implementation of the concurrent consumer/producer queue Q, how to perform local moves in `FindMoves`, and how to update the global data structures in `ApplyMoves`.

Algorithmus 4.4: Parallel Localized k-way Multi-try Local Search.

```
Input: Graph G = (V, E)
1  while True do
2      Queue Q = {v ∈ V : v is a boundary vertex}
3      mark all vertices as not moved
4      totalGain = 0
5      while Q is not empty do                          // start new global iteration
6          while Q is not empty do in parallel          // start new local iteration
7              v = Q.pop()
8              if v is moved then continue
9              V_pq = v ∪ {w ∈ N(v) : w is not moved}
               // priority queue with gain as key
10             PQ = {(gain(w), w) : w ∈ V_pq}
               // try to move boundary vertices
11             FindMoves(G, PQ)
12             If at least one PE stopped then signal other PEs to stop
13         Q, gain = ApplyMoves(G)
14         totalGain += gain
15         if the local quantile-based stopping rule signals to stop or gain = 0 then
16             break                                    // Stop current global iteration
17     if the global quantile-based stopping rule signals to stop or totalGain = 0 then
18         break                                        // Stop LMLS
```

Boundary Vertices and Concurrent Consumer/Producer Queue

We store the set of boundary vertices using a concurrent hash table by Maier et al. [MSD19]. The construction of the set is straightforward. We iterate over all vertices in parallel and if a vertex is at the boundary of a block then we insert it into the hash table.

We use a *concurrent consumer/producer queue* to decrease the probability that several PEs try to move the same vertex concurrently, that is, we want PEs to perform the local search in parts that share as few vertices as possible. In order to do this, the concurrent consumer/producer queue has p buckets that contain random subsets of boundary vertices and each bucket corresponds to one PE. To initialize the buckets (consuming phase), each PE iterates over the concurrent hash table and copies a random subset of the boundary vertices to its corresponding bucket. Note that there is no need to shuffle vertices in the buckets since the vertices in the hash table are already in random order. After the initialization step, a PE extracts a vertex from its

bucket (producing phase). If its bucket is empty then it tries to extract a vertex from the bucket of the next PE (circularly ordered). This is repeated until a non-empty bucket is found. Then a boundary vertex from this bucket is returned and the index of the found non-empty bucket is stored for the next step. Note that although it is possible to have high contention on the vertices of the same bucket, it is unlikely to be a bottleneck since a lot of work corresponds to each vertex. Specifically, all PEs perform local searches around extracted vertices and the running times of local searches are different and significantly greater than that of the extracting operation.

Finding moves (`FindMoves`)

Starting from a single boundary vertex, each PE moves vertices to find a sequence of moves that decreases the cut. However, all moves are local; that is, they do not affect the current global partition – moves are stored in the local memory of the PE performing them. To perform a move, a PE chooses a vertex with maximum gain and marks it so that other PEs cannot move it. Then, it updates the sizes of the affected blocks and saves the move. During the course of the algorithm, we store the sequence of moves yielding the best cut. We stop if there are no moves to perform or the adaptive stopping rule by Osipov and Sanders [OS10] signals the algorithm to stop (see details in Section 3.5.2). When a PE finished, the sequence of moves yielding the largest decrease in the edge cut is returned.

Implementation Details of `FindMoves`

In order to improve scalability, only the array for marking moved vertices is global. Note that within a local iteration, bits in this array are only ever set (using CAS) and never unset. Hence, the marking operation can be seen as priority update operation (see Shun et al. [Shu+13]) and thus causes only little contention. The algorithm keeps a local array of block sizes, a local priority queue, and a local hash table storing changed block IDs of vertices for each PE. Note that since the local hash table is small, it often fits into cache which is crucial for parallelization due to memory bandwidth limits. When the call to `FindMoves` finishes and the thread executing it notices that the queue Q is empty, it sets a global variable to signal the other PEs to finish the current call of the function `FindMoves`.

Let each PE process a set of edges $\mathcal{E}$ and a set of vertices $\mathcal{V}$. Each vertex can be moved only by one PE and moving a vertex requires the gain computation of its neighbors. Therefore, the parallel time of the function `FindMoves` is $\mathcal{O}(\sum_{v\in\mathcal{V}}\sum_{u\in N(v)} d(u) + |\mathcal{V}|) = \mathcal{O}(\sum_{v\in\mathcal{V}} d^2(v) + |\mathcal{V}|)$ since the gain of a vertex v can be updated at most $d(v)$ times. Note that we recalculate the gain of a vertex v from scratch after the move of its neighbor. The reason is that the block to which v has the strongest connection may change and we need to find a new block with the strongest connection. Note

that this is a pessimistic bound and it is possible to implement this function with $\mathcal{O}(|\mathcal{E}| \log \min(k, \Delta) + |\mathcal{V}|)$ parallel time. More precisely, for each vertex v we maintain a priority queue of size $\min(k', d(v))$ that returns a new block with the strongest connection in $\mathcal{O}(\log \min(k', d(v)))$ time.* Here k' is the number of blocks adjacent to v. Nevertheless, in our experiments, we recalculate gains from scratch since it requires less memory. Moreover, most of the vertices have relatively small degrees (especially in power-law graphs) and all implementations of priority queues have larger constants hidden in the running time bounds than scanning of neighbors of a vertex.

Applying Moves (`ApplyMoves`)

Let $M_i = \{B_{i1}, \dots\}$ denote the set of sequences of local moves found by PE i, where B_{ij} is a set of local moves performed by the j-th call of `FindMoves`. We apply moves sequentially in the order $M_1, M_2, \dots, M_p$. We cannot apply the moves directly in parallel since a move done by one PE can affect a move done by another PE. More precisely, assume that we want to move a vertex $v \in B_{ij}$ but we have already moved its neighbor w and this move of w was found by a different PE. Since the PE only knows its local changes, it calculates the gain of moving v (in `FindMoves`) according to the old block ID of w. If we then apply the rest of the moves in B_{ij} it may even increase the cut. To prevent this, we recalculate the gain of each move in a given sequence and remember the best cut. If there are no affected moves, we apply all moves from the sequence. Otherwise we apply only the part of the moves that gives the best cut with respect to the correct gain values. Finally, we insert all moved vertices into the queue Q. Let M be the set of all moved vertices during this procedure. The overall running time is then given by $\mathcal{O}(\sum_{v \in M} d(v))$. Note that if an initial partitioning algorithm generates balanced solutions then our parallel local search algorithm maintains balanced solutions, i.e. the balance constraint of our solution is never violated, since moves are applied sequentially.

4.3.4 Differences to Mt-Metis

We now discuss the differences between our algorithm and Mt-Metis. In the coarsening phase, our framework uses either the clustering or the matching algorithm while Metis is using only the matching algorithm. Our approach is especially well suited for networks that have a pronounced and hierarchical cluster structure. For example, in networks that contain star-like structures, a matching-based algorithm for coarsening matches only a single edge within these structures and hence cannot shrink the graph effectively. Moreover, it may contract "wrong" edges such as bridges. Using a clustering-based scheme, however, allows to contract the input graph more quickly in the multi-level scheme [MSS17]. The general initial partitioning scheme is similar in

*By using an array instead of a priority queue, we can achieve $\mathcal{O}(|\mathcal{E}| \min(k, \Delta) + |\mathcal{V}|)$ parallel running time since a new block with the strongest connection can be found in $\mathcal{O}(\min(k', d(v)))$ time

both algorithms. However, the employed sequential techniques differ because different sequential tools (KaHIP and Metis) are used to partition the coarsest graphs. In terms of local search, unlike Mt-Metis, our approach guarantees that the updated partition is balanced if the input partition is balanced and that the cut can only decrease or stay the same. The hill-climbing technique of `Mt-Metis`, however, may increase the cut of the input partition or may compute an imbalanced partition even if the input partition is balanced. Our algorithm has these guarantees since each PE performs moves of vertices locally in parallel. When all PEs finish, one PE globally applies the best sequences of local moves computed by all PEs. Usually, the number of applied moves is significantly smaller than the number of the local moves performed by all PEs, especially on large graphs. Thus, the main work is still made in parallel. Additionally, in the following section we introduce a cache-aware hash table that we use to store local changes of block IDs made by each PE. This hash table is more compact than an array and takes the locality of data into account.

4.4 Further Optimizations

In this section, we describe further optimization techniques that we use to achieve better speed-ups and overall speed. More precisely, we use cache-aligned arrays to mitigate the problem of false-sharing, the TBB scalable allocator [Tbb] for concurrent memory allocations and pin threads to cores to avoid rescheduling overheads. Additionally, we use a cache-aware hash table which we now describe. In contrast to traditional hash tables, this hash table allows us to exploit locality of data and hence we expect it to reduce the overall running time of the algorithm. Furthermore, to mitigate negative effects of NUMA (non-uniform memory access), we use the NUMA policy library [Lib]. Specifically, we use round robin memory allocation that allocates memory uniformly on all NUMA nodes.

4.4.1 Cache-Aware Hash Table

The main goal here is to improve the performance of our algorithm on large graphs. For large graphs, the gain computation in the LMLS routine dominates the overall running time. Recall that computing the gain of a vertex requires a local hash table for each PE and, thus, we expect that using a cache-aware technique reduces the overall running time. A cache-aware hash table combines properties of both an array and a hash table. It tries to store data with similar integer keys within the neighboring cache lines, thus reducing the cost of subsequent accesses to these keys. On the other hand, it still consumes less memory than an array which is crucial for the hash table to fit into caches.

We implement a cache-aware hash table using the linear probing technique and tabulation hashing as hash function [PT11]. Linear probing typically outperforms

other collision resolution techniques in practice and the computation of the tabulation hash function can be done with a very little overhead. The tabulation hash function works as follows. Let $x = x_1 \dots x_k$ be a key to be hashed, where x_i represents t bits of the binary representation of x. Let $T_i, i \in [1, k]$ be tables of size 2^t, where each element is a random 32-bit integer. Using $\oplus$ as exclusive-or operation, the tabulation hash function is then defined as follows:

$$h(x) = T_1[x_1] \oplus \dots \oplus T_k[x_k].$$

Exploiting Locality of Data

As our experiments indicate, the distribution of keys that we access during the computation of the gains is not uniform. Instead, it is likely that the time between accesses to two consecutive keys is small. On typical systems currently used, the size of a cache line is often 64 bytes (16 elements with 4 bytes each). Now suppose our algorithm accesses 16 consecutive vertices one after another. If we would use an array storing the block IDs of all vertices instead of a hash table, we can access all block IDs of the vertices with only one cache miss. A hash table on the other hand does not give any locality guarantees. On the contrary, it is very probable that consecutive keys are hashed to completely different parts of the hash table. However, due to memory constraints we cannot use an array to store block IDs for each PE in the `FindMoves` procedure.

However, even if the arrays were to fit into memory, doing so would be problematic. To see this let $|L2|$ and $|L3|$ be the sizes of L2 and L3 caches of a given system, respectively, and let p' be the number of PEs used per NUMA node. For large graphs, the array may not fit into $\max(|L2|, |L3|/p')$ memory. In this case, each PE will access its own array in main memory which affects the running time due to the available memory bandwidth. Thus, we want a compact data structure that fits into $\max(|L2|, |L3|/p')$ memory most of the time *and* preserves the locality guarantees of an array.

For this, we modify the tabulation hash function from above according to Mehlhorn and Sanders [MS08]. More precisely, let $x = x_1 \dots x_{k-1} x_k$, where x_k are the l least significant bits of x and $x_1, \dots, x_{k-1}$ are t bits each. Then we compute the tabulation hash function as follows:

$$h(x) = T_1[x_1] \oplus \dots \oplus T_{k-1}[x_{k-1}] \oplus x_k.$$

This guarantees that if two keys x and x' differ only in first l bits and, hence, $|x - x'| < 2^l$ then $|h(x) - h(x')| < 2^l$. Thus, if $l = \mathcal{O}(\log c)$, where c is the size of a cache line, then x and x' are in the same cache line when accessed. This hash function introduces at most 2^l additional collisions since if we do not consider the l least significant bits of a key then at most 2^l keys have the same remaining bits. For our experiments, we chose $k = 3, l = 5$, and $t = 10$.

4.5 Experimental Evaluation

In this section, we present a detailed experimental analysis of our shared-memory graph partitioning framework and its main competitors. First, in Section 4.5.1, we discuss the methodology of the experiments as well as the machines and graphs used in the experiments. Next, we compare quality and performance of our algorithm against state-of-the-art competitors in Sections 4.5.2 and 4.5.3, respectively. Moreover, in Section 4.5.4, we present comparison in terms of memory consumption, which is an important for processing large graphs. Finally, we show experimental analysis of different parts of our algorithm in Section 4.5.5. This analysis shows that each part of the algorithm is crucial to achieve good speed-ups and quality of the partitioning.

4.5.1 Methodology

Our framework `Mt-KaHIP` (Multi-threaded Karlsruhe High Quality Partitioning) is based on the sequential framework `KaHIP` [MSS14; SS11] and implemented using `C++` and the `C++17` multi-threading library. All binaries are built using `g++-7.3.0` with the `-O3` flag and 64-bit index data types. We perform all experiments using machines A (four sockets) and B (two sockets) (see Section 2.3.4 for details). The reason behind this is to investigate the dependence between performance and NUMA effects. Specifically, we evaluate all experiments using $1, 40$, and 79 PEs on machine A and $1, 16$, and 31 on machine B. Note that we always leave at least one PE for the operating system.

We compare ourselves to `Mt-Metis 0.6.0` using the default configuration with hill-climbing enabled (`Mt-Metis`) as well as sequential `KaHIP 2.0` using the *fast social* configuration (`KaHIP`) and `ParHIP 2.0` [MSS17] using the *fast social* configuration (`ParHIP`). According to LaSalle and Karypis [LK13] `Mt-Metis` has better speed-ups and running times compared to `ParMetis` [KK99] and `Pt-Scotch` [Pel12]. At the same time, it produces partitions of similar quality. Furthermore, `ParHIP` [MSS17] achieves better quality and performance than `ParMetis`, `PT-Scotch`, `PuLP` [SMR14], and the distributed graph partitioning framework by Uganer and Backstrom [UB13]. Therefore, we do not perform experiments with `ParMetis`, `Pt-Scotch` and the distributed graph partitioning framework by Uganer and Backstrom to save running time and to save running time and to keep the experimental evaluation simple. However, in order to have a more complete evaluation we additionally compare `Mt-KaHIP` to `DiBaP` [Mey12; MMS09a] and `PuLP`. We present the comparisons in separate sections to keep the experimental evaluation simple and due to inability to run `DiBaP` on our benchmark set of instances.

Algorithm Configuration. Any multi-level algorithm has a considerable number of choices between algorithmic components and tuning parameters. We adopt parameters from the coarsening and initial partitioning phases of `KaHIP` (fast social configuration).

The default `Mt-KaHIP` configuration uses 10 and 25 label propagation iterations during coarsening and refinement, respectively. It partitions a coarse graph $\max(p, 4)$ times during the initial partitioning phase. We use a stopping rule during the coarsening phase that signals to stop when the coarsest graph has less than $5000 \cdot k$ vertices or the contraction ratio is less than 1.1. Here the contraction ratio is the ratio between the number of vertices in the graph before and after contraction. Furthermore, we compute 90% quantiles in *global* stopping rules of LMLS which is used to decide whether to perform a new global iteration or not. The same quantiles are used in the *local* stopping rule of LMLS for local iterations.

The `Mt-KaHIP fast` configuration uses a more aggressive stopping rule during the coarsening phase. The reason for this is that when the coarsening phase finishes in `Mt-KaHIP`, it may occur that the coarsest graph is not small enough. As a result, the initial partitioning phase runs considerably faster than other phases and, thus, initial partitioning becomes a bottleneck (see Section 4.5.5). Therefore, the stopping rule in `Mt-KaHIP fast` uses the stopping rule used in `Mt-KaHIP` but allows to contract graphs further. Specifically, every time the stopping rule in `Mt-KaHIP` would have stopped coarsening, `Mt-KaHIP fast` continues to coarsen the graph while the maximum number of attempts (2) is not exceeded and the coarsest graph has more than 0.1% edges of the input graph or more than $300K$ edges.

The `Mt-KaHIP eco` configuration uses the parallel local max matching algorithm (see Section 4.3.1) to construct a matching and uses it to contract the graph during the coarsening phase. We developed this configuration since the parallel label propagation algorithm may compute partition of bad quality on the mesh type graph that do not have communities since the parallel label propagation algorithm tries to assign communities to the same clusters. Therefore, if the input graph does not have communities it is possible that the resulting partition does not have good quality. However, matching algorithms are not biased towards communities. We use a combination of stopping rules of `Mt-KaHIP` and `Mt-KaHIP fast`. More precisely, when we contract the graph using the parallel local matching algorithm, we use the the same stopping rules as in `Mt-KaHIP`. When this stopping rule signals to stop, we start to use the parallel label propagation algorithm and the stopping rule of `Mt-KaHIP fast` to contract the coarsest graph even further.

Our default value of allowed imbalance is 3% – this is one of the values used in [WC00a] and in `Mt-Metis`. We call a solution imbalanced if at least one block exceeds this amount. By default, we perform ten repetitions for every algorithm using different random seeds for initialization and report the arithmetic average of computed cut size and running time on a per instance (graph and number of blocks k) basis. If at least one repetition returns an imbalanced partition of an instance then we mark this instance imbalanced. Table 4.3 shows the number of instances partitioned with imbalance by different frameworks. When further averaging over multiple instances, we use the *geometric mean* for cut sizes and running times and the *harmonic mean* for the relative speed-ups in order to give every instance a comparable influence on

the final score. Table 4.3 shows the number of imbalanced partitions constructed by different frameworks. Our experiments focus on the cases $k \in \{16, 64\}$ to save running time and to keep the experimental evaluation simple. These values of k are sufficient to evaluate performance and quality of the frameworks.

Table 4.3: Number of instances partitioned without imbalance by each algorithm on machines A and B. Note that we did not run `Mt-KaHIP fast` and `Mt-KaHIP eco` on machine B. We did not run `Mt-KaHIP eco` with $p = 40$ on machine A.

Algorithm	p	Machine A	p	Machine B
`Mt-KaHIP`	1	0	1	0
	40	0	16	0
	79	0	31	0
`Mt-KaHIP fast`	1	1	1	–
	40	1	16	–
	79	1	31	–
`Mt-KaHIP eco`	1	0	1	–
	40	–	16	–
	79	0	31	–
`ParHIP`	1	2	1	1
	40	10	16	11
	79	12	31	10
`Mt-Metis`	1	2	1	2
	40	48	16	47
	79	47	31	47
`KaHIP`	1	0	1	0
`PuLP`	1	18	1	10
	40	28	16	21
	79	32	31	20

Instances. We perform experiments on all graphs shown in Table 2.3.2 excluding the large graphs twitter-2010, clueweb12, uk-2014, and eu-2015 to save running time. Therefore, our benchmark consists of 38 graphs and 76 instances. We denote the set of all instances as $\mathcal{I}$.

Note that only `Mt-KaHIP` and `ParHIP` with $p = 1, 40, 79$ were able to partition all instances on machine A, whereas only `Mt-KaHIP` with $p = 1, 16, 31$ and `ParHIP` with $p = 1, 16$ were able to partition all instances on machine B. Therefore, in order to perform an extensive comparison of the frameworks on different sets of instances, we consider multiple subsets of the instances $\mathcal{I}$.

Specifically, we consider the following four subsets of the instances $\mathcal{I}$ on machine A:

- Instances where `Mt-KaHIP`, `Mt-KaHIP fast`, `Mt-Metis`, `KaHIP`, and `ParHIP` succeeded: $\mathcal{S}_{\texttt{All}} = \mathcal{S}_{\texttt{Mt-Metis}} \cap \mathcal{S}_{\texttt{KaHIP}} \cap \mathcal{S}_{\texttt{Mt-Metis},p=1}$ and $|\mathcal{S}_{\texttt{All}}| = 67$.
- Instances where `Mt-Metis` with $p = 40, 79$ succeeded:

$$\begin{aligned}\mathcal{S}_{\texttt{Mt-Metis}} = \mathcal{I} \setminus \{&(\text{ba\_2\_22}, k = 64), (\text{er\_2\_22\_2\_23}, k = 64),\\ &(\text{sk-2005}, k = 16), (\text{sk-2005}, k = 64),\\ &(\text{uk-2007}, k = 16), (\text{uk-2007}, k = 64)\}\end{aligned}$$

  and $|\mathcal{S}_{\texttt{Mt-Metis}}| = 70$.
- Instances where `KaHIP` succeeded:

$$\begin{aligned}\mathcal{S}_{\texttt{KaHIP}} = \mathcal{I} \setminus \{&(\text{rgg\_2\_27}, k = 16), (\text{rgg\_2\_27}, k = 64),\\ &(\text{sk-2005}, k = 16), (\text{sk-2005}, k = 64),\\ &(\text{uk-2007}, k = 16), (\text{uk-2007}, k = 64)\}\end{aligned}$$

  and $|\mathcal{S}_{\texttt{KaHIP}}| = 70$.
- Instances where `Mt-Metis` with a single PE succeeded:

$$\mathcal{S}_{\texttt{Mt-Metis},p=1} = \mathcal{I} \setminus \{(\text{uk-2005}, k = 64), (\text{uk-2007}, k = 16), (\text{uk-2007}, k = 64)\}$$

  and $|\mathcal{S}_{\texttt{Mt-Metis},p=1}| = 73$.
- Instances where `PuLP` with $p = 1, 40, 79$ succeeded:

$$\begin{aligned}\mathcal{S}_{\texttt{PuLP}} = \mathcal{I} \setminus \{&(\text{amazon}, k = 16), (\text{amazon}, k = 64),\\ &(\text{youtube}, k = 16), (\text{youtube}, k = 64)\}\end{aligned}$$

  and $|\mathcal{S}_{\texttt{PuLP}}| = 72$.

We only present comparisons of speed-ups and running times of the frameworks on machine B since partitioning quality is machine independent. However, quality of partitions computed by `Mt-KaHIP` depends on the number of PEs. We consider the following two subsets of instances $\mathcal{I}$ that were partitioned by `Mt-Metis` and `ParHIP` on machine B:

- Instances where `Mt-KaHIP`, `Mt-Metis`, and and `ParHIP` with $p = 31$ succeeded: $\mathcal{S}^B_{\texttt{All}} = \mathcal{S}^B_{\texttt{Mt-Metis}} \cap \mathcal{S}^B_{\texttt{ParHIP}}$ and $|\mathcal{S}^B_{\texttt{All}}| = 68$.
- Instances where `Mt-Metis` with $p = 16, 31$ succeeded: $\mathcal{S}^B_{\texttt{Mt-Metis}}$ – the same set as on machine A.
- Instances where `ParHIP` with 31 PEs succeeded:

$$\mathcal{S}^B_{\texttt{ParHIP}} = \mathcal{I} \setminus \{(\text{rhg\_2\_23\_d\_8}, k = 16), (\text{rhg\_2\_23\_d\_8}, k = 64)\}$$

  and $|\mathcal{S}^B_{\texttt{ParHIP}}| = 74$.

4.5.2 Quality

In this section, we compare our algorithm against competing state-of-the-art algorithms in terms of quality. Specifically, we compare ourselves to `Mt-Metis`, `ParHIP`, and `KaHIP`. Further in this section, we additionally compare `Mt-KaHIP` to `DiBaP`, `PuLP`, and `Mt-KaHIP eco`. All the experiments for quality comparison were performed on machine A since partitioning quality is machine independent. However, quality of partitions computed by `Mt-KaHIP` depends on the number of PEs.

The performance plots in Figure 4.4 (see Section 2.3.1) show the results of our experiments. We use the symmetric log scaling on the y-axis which allows to define a range around zero that is scaled linearly, whereas the rest of the y-axis is scaled logarithmically. In this performance plot, we have a linear scaling up to 0.05 since most of the points are concentrated below 0.05. Our algorithm gives the best overall quality, usually producing the best cut for the most instances. Even for the small fraction of instances where other algorithms are better, our algorithm is at most 7.8% worse. The overall solution quality does not heavily depend on the number of PEs used. In particular, more PEs result in a slightly higher partitioning quality since more initial partition attempts are done in parallel. The original fast social configuration of `KaHIP` as well as `ParHIP` produce worse quality than `Mt-KaHIP`. This is due to the high-quality local search scheme that we use; i.e., parallel LMLS significantly improves solution quality. `Mt-Metis` with $p = 1$ has worse quality than our algorithm on most instances except 23 mesh type networks and one complex network. For `Mt-Metis` this is expected since it is considerably faster than our algorithm. However, `Mt-Metis` also suffers from deteriorating quality and many imbalanced partitions as the number of PEs goes up. This is mostly the case for complex networks. For example, `Mt-Metis` with 79 PEs produces imbalanced partitions for 67.1% of the instances. Table 4.4 shows the geometric means of the cut sizes over all instances, including imbalanced solutions. For $p = 79$, the geometric mean cut size of `Mt-KaHIP` is 14.7% smaller than that of `Mt-Metis` on the set of instances $\mathcal{S}_{\texttt{Mt-Metis}}$. Furthermore, the geometric mean cut size of `Mt-KaHIP` with 79 PEs is 12.5% smaller than that of `ParHIP` with 79 PEs on the set of instances $\mathcal{I}$ and 7.1% smaller than that of `KaHIP` on the set of instances $\mathcal{S}_{\texttt{Mt-Metis}}$. Moreover, the Wilcoxon signed-rank test (see Section 2.3.3) indicates that the quality advantage of `Mt-KaHIP` over the other solvers is statistically significant. Table 4.5 shows the relative difference ($|a - b| / \max(a, b)$) of the geometric mean cut sizes and p-values.

Additionally, we present the performance plot in Figure 4.5 that compares `Mt-KaHIP fast` with $p = 79$ to `Mt-Metis` and `ParHIP`. To have a fair quality comparison of `Mt-Metis` and `Mt-KaHIP fast`, we only present the results of `Mt-KaHIP fast` with $p = 79$ since some instances are partitioned better by `Mt-KaHIP fast` with $p = 1, 40$ than by `Mt-KaHIP fast` with $p = 79$. `Mt-KaHIP fast` with $p = 79$ has better partitioning quality than `ParHIP` with $p = 79$ and `KaHIP`. For $p = 79$, the geometric mean cut size of `Mt-KaHIP fast` is 10.4% smaller than that of `Mt-Metis` on the set

of instances $\mathcal{S}_{\texttt{Mt-Metis}}$. Furthermore, the geometric mean cut size of `Mt-KaHIP` with 79 PEs is 7.9% smaller than that of `ParHIP` with 79 PEs on the set of instances $\mathcal{I}$ and 2.4% smaller than that of `KaHIP` on the set of instances $\mathcal{S}_{\texttt{KaHIP}}$. Moreover, the Wilcoxon signed-rank test (see Section 2.3.3) indicates that the quality advantage of `Mt-KaHIP fast` over the other solvers *except* `Mt-Metis` is statistically significant. However, note that the effectiveness tests from the next paragraph show that `Mt-KaHIP fast` with $p = 79$ has better quality than `Mt-Metis` with $p = 79$. Furthermore, if we consider only instances that are partitioned without imbalance by `Mt-Metis` then the p-value is 10^{-5} and, thus, the difference is statistically significant. However, note that there are only 23 instances that are partitioned without imbalance by `Mt-Metis` which may not be enough to show that the difference is statistically significant.

Figure 4.6 show the performance plots that compares `Mt-KaHIP` and `Mt-KaHIP fast` with $p = 79$. It is not surprising that `Mt-KaHIP` has better quality than `Mt-KaHIP fast` since `Mt-KaHIP fast` uses a more aggressive coarsening scheme. `Mt-KaHIP` produces better partitions for 63 instances. Furthermore, `Mt-KaHIP` has a 18.8% smaller cut than `Mt-KaHIP fast` in the best case and only a 2.2% larger cut than `Mt-KaHIP fast` in the worst case.

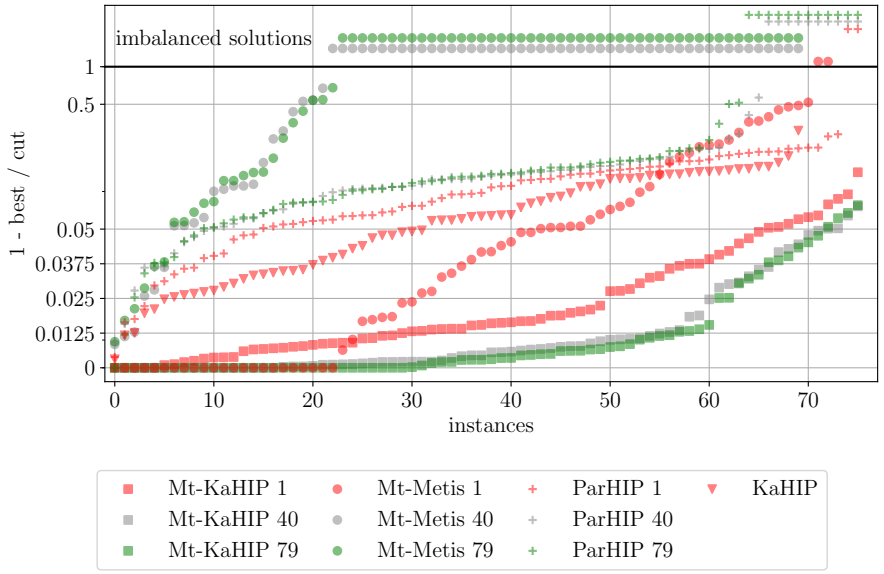


Figure 4.4: Performance plot for the cut size of `Mt-KaHIP` and competitors. The number behind the algorithm name denotes the number of PEs used.

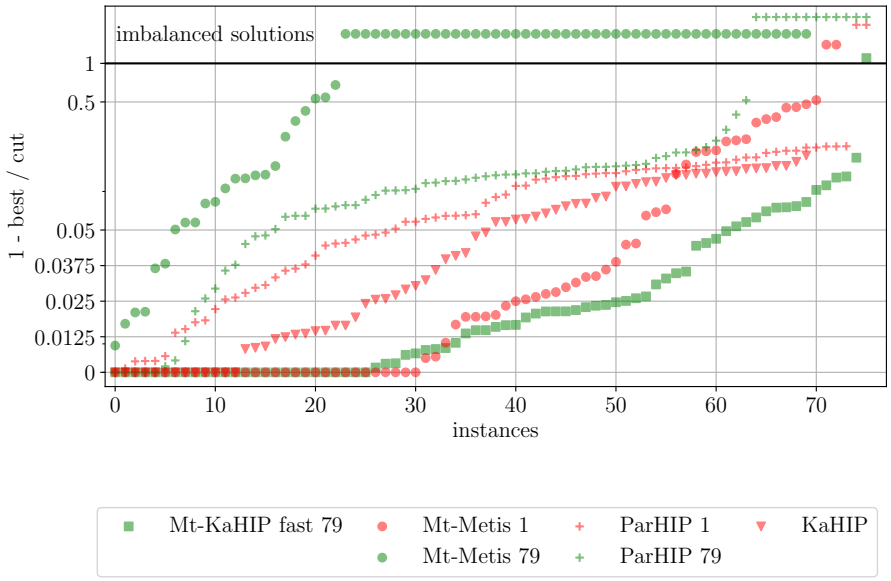


Figure 4.5: Performance plot for the cut size of `Mt-KaHIP fast` and competitors.

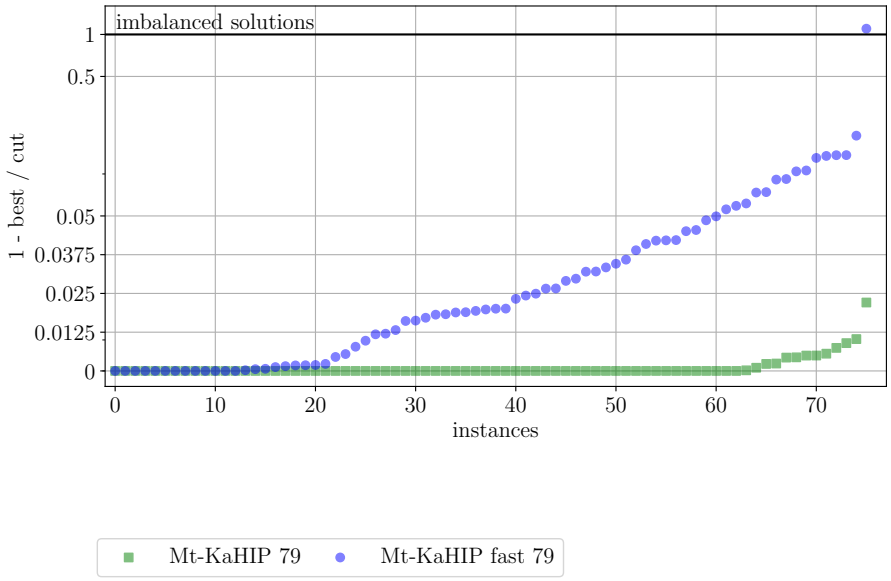


Figure 4.6: Performance plot for the cut size of `Mt-KaHIP` and `Mt-KaHIP fast`.

Table 4.4: Geometrical means of cut sizes for different frameworks evaluated on different sets of instances.

		Instances				
Algorithm	p	$\mathcal{S}_{\texttt{All}}$	$\mathcal{S}_{\texttt{Mt-Metis}}$	$\mathcal{S}_{\texttt{KaHIP}}$	$\mathcal{S}_{\texttt{Mt-Metis},p=1}$	$\mathcal{I}$
Mt-KaHIP	1	1593.6K	1620.9K	1716.3K	1718.6K	1846.9K
	40	1574.5K	1601.3K	1696.6K	1697.6K	1824.4K
	79	**1571.5K**	**1598.0K**	**1693.2K**	**1695.8K**	**1822.4K**
Mt-KaHIP fast	1	1664.7K	1690.7K	1790.2K	1800.0K	1934.1K
	40	1655.6K	1681.3K	1781.4K	1786.3K	1921.2K
	79	1652.3K	1678.4K	1778.2K	1783.4K	1916.6K
ParHIP	1	1749.7K	1794.1K	1889.2K	1905.2K	2045.8K
	40	1851.0K	1800.3K	1899.0K	1934.3K	2075.6K
	79	1853.2K	1803.8K	1902.7K	1939.5K	2081.5K
Mt-Metis	1	1759.9K	–	–	1876.0K	–
	40	1768.5K	1871.2K	–	–	–
	79	1772.1K	1873.3K	–	–	–
KaHIP	1	1695.0K	–	1822.0K	–	–

Table 4.5: Pairwise comparison of Mt-KaHIP and Mt-KaHIP fast for $p = 1, 40, 79$ to other competitors. We compare Mt-KaHIP and Mt-KaHIP fast against competitors on the largest set of instances which were partitioned by competitors. For example, for Mt-KaHIP and Mt-Metis with $p = 40, 79$ this is set $\mathcal{S}_{\texttt{Mt-Metis}}$. Each cell of the table is the relative difference of the geometric mean cut sizes and p-values.

Algorithm		Mt-Metis			ParHIP			KaHIP
	p	1	40	79	1	40	79	1
Mt-KaHIP	1	8.4% $8 \cdot 10^{-3}$	13.4% $3 \cdot 10^{-4}$	13.5% $5 \cdot 10^{-5}$	9.7% $4 \cdot 10^{-14}$	11.0% $4 \cdot 10^{-14}$	11.3% $3 \cdot 10^{-13}$	5.8% $3 \cdot 10^{-12}$
	40	9.5% $2 \cdot 10^{-3}$	14.4% $7 \cdot 10^{-5}$	14.5% $2 \cdot 10^{-5}$	10.8% $7 \cdot 10^{-14}$	12.1% $4 \cdot 10^{-14}$	12.4% $7 \cdot 3^{-13}$	6.9% $7 \cdot 6^{-13}$
	79	9.6% $2 \cdot 10^{-3}$	14.6% $6 \cdot 10^{-5}$	14.7% $1 \cdot 10^{-5}$	10.9% $7 \cdot 10^{-14}$	12.2% $4 \cdot 10^{-14}$	12.5% $3 \cdot 10^{-13}$	7.1% $7 \cdot 10^{-13}$
Mt-KaHIP fast	1	4.0% 0.9	9.6% 0.3	9.8% 0.1	5.5% $9 \cdot 10^{-7}$	6.8% $3 \cdot 10^{-10}$	7.1% $1 \cdot 10^{-9}$	1.7% $4 \cdot 10^{-3}$
	40	4.8% 0.5	10.1% 0.2	10.3% 0.06	6.1% $4 \cdot 10^{-6}$	7.4% $1 \cdot 10^{-10}$	7.7% $7 \cdot 10^{-10}$	2.2% $3 \cdot 10^{-3}$
	79	4.9% 0.5	10.2% 0.1	10.4% 0.07	6.3% $2 \cdot 10^{-6}$	7.7% $2 \cdot 10^{-10}$	7.9% $9 \cdot 10^{-10}$	2.4% $3 \cdot 10^{-3}$

Effectiveness Tests. The idea of effectiveness tests is to give the faster algorithm the same amount of time as the slower algorithm for additional repetitions that can lead to improved solutions. We have improved an approach used in [SS11] to extract more information out of a moderate number of measurements. Suppose we want to compare two algorithms A and B and that we run k times on an instance I (graph and number of blocks). We generate a *virtual instance* as follows. We first sample one of the repetitions of both algorithms. Let t_1^A and t_1^B refer to the observed running times. Without loss of generality assume $t_1^A \geq t_1^B$. Now we sample (without replacement) additional repetitions of algorithm B until the total running time accumulated for algorithm B exceeds t_1^A. We assume that there are always enough repetitions of algorithm B for that. More generally, let t_ℓ^B denote the running time of the last sample. We accept the last sample of B with probability $(t_1^A - \sum_{1 \leq i < \ell} t_i^B)/t_\ell^B$.

Theorem 4.1

The expected total running time of accepted samples for B is the same as the running time for the single repetition of A.

Proof. Let $t = \sum_{1<i<\ell} t_i^B$, where $\ell > 1$. Consider a random variable T that is the total time of sampled repetitions. With probability $p = (t_1^A - t)/t_\ell^B$, we accept ℓ-th sample and with probability $1 - p$ we decline it. Then

$$\begin{aligned} \mathsf{E}[T] &= p \cdot (t + t_\ell^B) + (1 - p) \cdot t \\ &= \frac{t_1^A - t}{t_\ell^B} \cdot (t + t_\ell^B) + (1 - \frac{t_1^A - t}{t_\ell^B}) \cdot t = t_1^A \end{aligned} \tag{4.1}$$

□

The quality assumed for A in this virtual instance is the quality of the only run of algorithm A. The quality assumed for B is the best quality observed for an accepted sample for B.

It is possible that the running times of algorithms for one virtual instance differ significantly. We want to derive a probabilistic bound on the difference of the running times of slower and faster algorithms. We prove that if a specific number of virtual instances is generated then the total running time of faster and slower algorithms over all virtual instances is within a certain range. However, note that the algorithm A can be faster in one virtual instance, whereas it is slower in another virtual instance. Therefore, it is possible that the total running time over all instances of the algorithm A is still significantly differs from that of the algorithm B. However, it never happens in our effectiveness tests.

We will use the following theorem.

Theorem 4.2 (Hoeffding's inequality [Hoe63])

Let $X_1, \ldots, X_n$ be independent random variables with $\mathsf{E}[X_i] = \mu_i$ and $P(a_i \leq X_i \leq$

$b_i) = 1$ *for constants* a_i *and* b_i*. Then*

$$P(|\sum_{i=1}^{n} X_i - \sum_{i=1}^{n} \mu_i| \geq \epsilon) \leq 2 \cdot \mathrm{e}^{-2\epsilon^2 / \sum_{i=1}^{n} (b_i - a_i)^2}.$$

Let T^A and T^B denote sets that consist of running times of all k repetitions of algorithms A and B, respectively. Let us consider n virtual instances. Note that the same algorithm may be both slow for one instance and fast for another. This depends on the results of the sampling from repetitions. Furthermore, let t_i^S denote the running time of the first sample of the slower algorithm in the i-th virtual instance. Let t_{ij}^F denotes the running time of the j-th sample of the faster algorithm in the i-th virtual instance. Let f_i denotes the number of the samples of the faster algorithm in the i-th virtual instance. Let $t_{max} = \min\{\max T^A, \max T^B\}$ if $t^A \geq t^B$ or $t^A \leq t^B \ \forall t^A \in T^A$ and $\forall t^B \in T^B$ otherwise $t_{max} = \max(T^A \cup T^B)$. Let $t_{min} = \max\{\min T^A, \min T^B\}$. Thus, t_{max} is an upper bound on $t_{ij}^F \ \forall j \in [1, f_i]$ and t_{min} is a lower bound on t_i^S. Now we prove the following theorem.

Theorem 4.3
Consider a random variable $X_i = \sum_{j=1}^{f_i - 1} t_{ij}^F + t_{if_i}^F$*, where* $f_i > 1$*. We accept the* f_i*-th sample with probability* $(t_i^S - \sum_{j=1}^{f_i - 1} t_{ij}^F)/t_{if_i}^F$*. Let* $\mathsf{E}[X_i] = \mu_i$ *and* $\mu = \sum_{i=1}^{n} \mu_i$*. Then*

$$P(|\sum_{i=1}^{n} X_i - \sum_{i=1}^{n} \mu_i| \geq \delta\mu) \leq 2 \cdot \mathrm{e}^{-2 \cdot n \cdot \delta^2 \cdot r^2},$$

where $r = \frac{t_{min}}{t_{max}}$ *is a data dependent constant.*

Proof. Note that $a_i = \sum_{j=1}^{f_i - 1} t_{ij}^F \leq X_i \leq \sum_{j=1}^{f_i} t_{ij}^F = b_i$ and $X_1, \ldots, X_n$ are independent random variables since they correspond to different virtual instances. Here a_i and b_i are the parameters from Hoeffding's inequality. Then $t_i' = b_i - a_i = t_{if_i}^F$. Now we can apply Hoeffding's inequality, where $\epsilon = \delta\mu$. Then

$$\begin{aligned} P(|\sum_{i=1}^{n} X_i - \sum_{i=1}^{n} \mu_i| \geq \delta\mu) &\leq 2 \cdot \exp(-2\delta^2 (\sum_{i=1}^{n} \mu_i)^2 / \sum_{i=1}^{n} t_i'^2) \\ &= 2 \cdot \exp(-2\delta^2 (\sum_{i=1}^{n} t_i^S)^2 / \sum_{i=1}^{n} t_i'^2) \qquad (4.2) \\ &\leq 2 \cdot \exp(-2\delta^2 (n \cdot t_{min})^2 / (n \cdot t_{max}^2)) \\ &\leq 2 \cdot \mathrm{e}^{-2\delta^2 n r^2} \end{aligned}$$

□

Corollary 4.4
If we want $P(|\sum_{i=1}^{n} X_i - \sum_{i=1}^{n} \mu_i| \geq \delta\mu) \leq \epsilon$ *then* $n \geq \ln \frac{2}{\epsilon} / (2\delta^2 r^2)$.

Although this formula gives a necessary number of virtual instances to use such that time difference between algorithms is within a desired percent, this number is different for each instance. Therefore, in order to be fair to all instances we consider 100 of virtual instances for each instance. This number is already large enough such that running time difference between algorithms is within 5% for almost all instances.

Figures 4.7 – 4.11 present performance plots with effectiveness test for `Mt-KaHIP` and competitors with $p = 1, 79$. We use the symmetric log scaling on the y-axis which allows to define a range around zero that is scaled linearly, whereas the rest of the y-axis is scaled logarithmically. In this performance plot, we have a linear scaling up to 0.05 since most of the points are concentrated below 0.05. Note that we compare our frameworks to competitors with $p = 1$ to show that `Mt-KaHIP` and `Mt-KaHIP fast` almost always have better quality than our competitors with $p = 1$ which produce better partitions than when using $p = 79$. Furthermore, competitors with $p = 1$ almost always produces balanced partitions unlike with $p = 79$. As we can see, even with additional running time other frameworks have mostly worse quality than `Mt-KaHIP` with $p = 1, 79$.

Table 4.6 shows the detailed information about the effectiveness tests. For $p = 79$, we can see that `Mt-KaHIP` always produces better cuts for 90.5%, 98.6%, and 99.5% of virtual instances against `Mt-Metis`, `ParHIP`, and `KaHIP`, respectively. `Mt-KaHIP fast` with $p = 79$ always produces better cuts for 88.4%, 91.7%, and 82.4% of virtual instances against `Mt-Metis` with $p = 79$, `ParHIP` with $p = 79$, and `KaHIP`, respectively. Furthermore, `Mt-KaHIP` with $p = 79$ has a 70.3% smaller cut in the best case and a 7.9% larger cut in the worst case. `Mt-KaHIP fast` with $p = 79$ has a 72.4% smaller cut in the best case and a 87.7% larger cut in the worst case. Note that the high relative differences in the worst cases correspond to imbalanced partitions of the graph in-2004 produced by `Mt-KaHIP fast`. If we consider only virtual instances partitioned without imbalance then the relative differences are not that high.

Table 4.6: Results of effectiveness tests. Here "% of instances" is the percent of virtual instances partitioned better by our framework. "Mean %" is the relative difference ($|a-b|/\max(a,b)$) between the geometric mean cut sizes computed by our framework and a competitor. If a value is preceded by a sign $-$ than our algorithms has smaller geometric mean cut sizes, otherwise there is a sign $+$. "Best %" is the *best* relative difference between the cut size computed by our framework and a competitor. "Worst %" is the *worst* relative difference between the cut size computed by our framework and a competitor.

Algorithm			Mt-Metis		ParHIP		KaHIP
	p		1	79	1	79	1
Mt-KaHIP	1	% of instances	57.7	84.7	97.1	93.6	89.5
		Mean %	−7.2	−9.5	−9.9	−8.5	−5.2
		p-value	$1 \cdot 10^{-88}$	$1 \cdot 10^{-36}$	0	0	0
		Best %	53.8	59.9	57.0	50.0	44.5
		Worst %	18.6	6.3	27.7	24.6	33.7
	79	% of instances	69.9	90.5	99.2	98.6	99.5
		Mean %	−10.4	−12.7	−12.1	−12.3	−8.0
		p-value	$7 \cdot 10^{-265}$	$2 \cdot 10^{-170}$	0	0	0
		Best %	54.4	70.3	57.7	60.7	45.5
		Worst %	7.6	6.5	1.5	7.9	4.8
Mt-KaHIP fast	1	% of instances	46.3	80.4	82.2	80.7	65.5
		Mean %	−3.1	−5.7	−6.0	−4.4	−1.3
		p-value	$8 \cdot 10^{-3}$	$8 \cdot 10^{-3}$	0	0	$9 \cdot 10^{-101}$
		Best %	53.2	59.8	34.2	41.2	19.3
		Worst %	78.2	6.4	80.0	81.6	83.9
	79	% of instances	61.5	88.4	91.4	91.7	82.4
		Mean %	−7.0	−9.8	−9.4	−9.0	−5.1
		p-value	$7 \cdot 10^{-93}$	$2 \cdot 10^{-82}$	0	0	0
		Best %	54.2	72.4	38.4	55.0	26.7
		Worst %	87.1	7.7	59.3	87.7	63.3

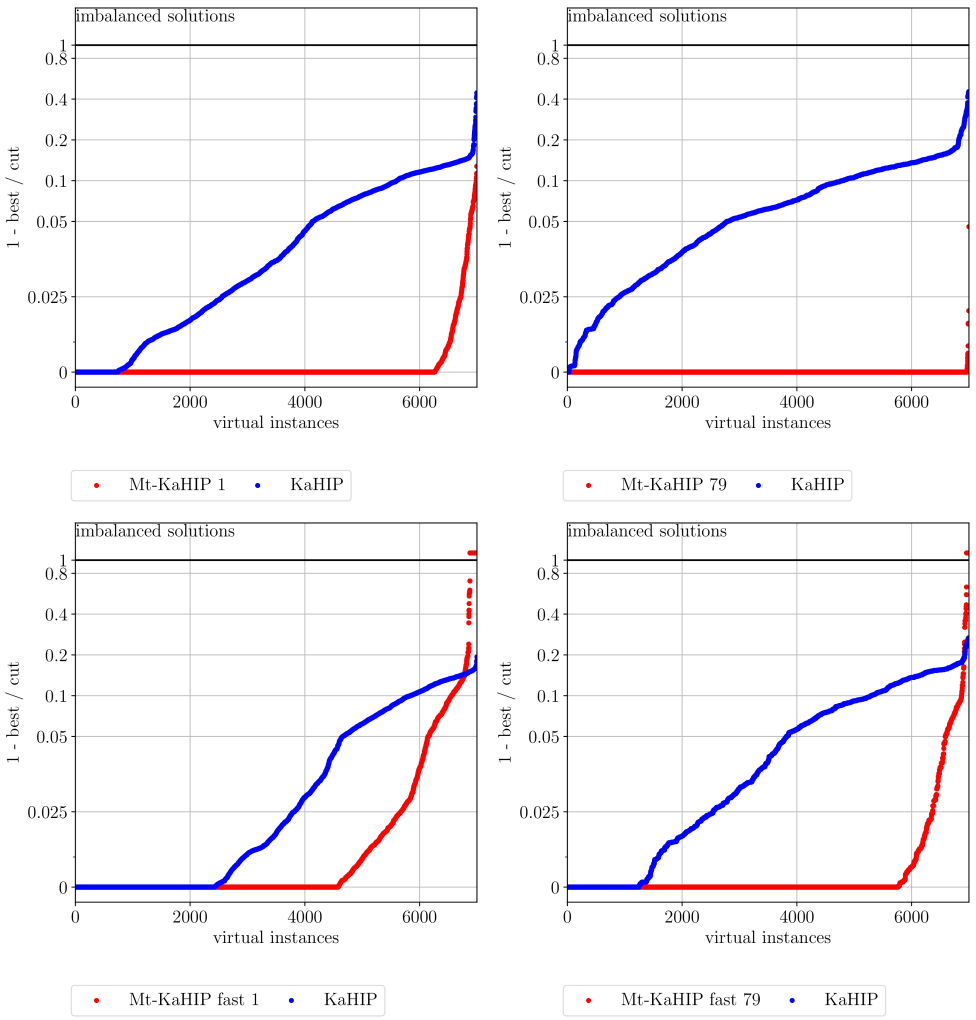


Figure 4.7: Effectiveness tests for `Mt-KaHIP`, `Mt-KaHIP fast` and `KaHIP`. The number behind the algorithm name denotes the number of PEs used.

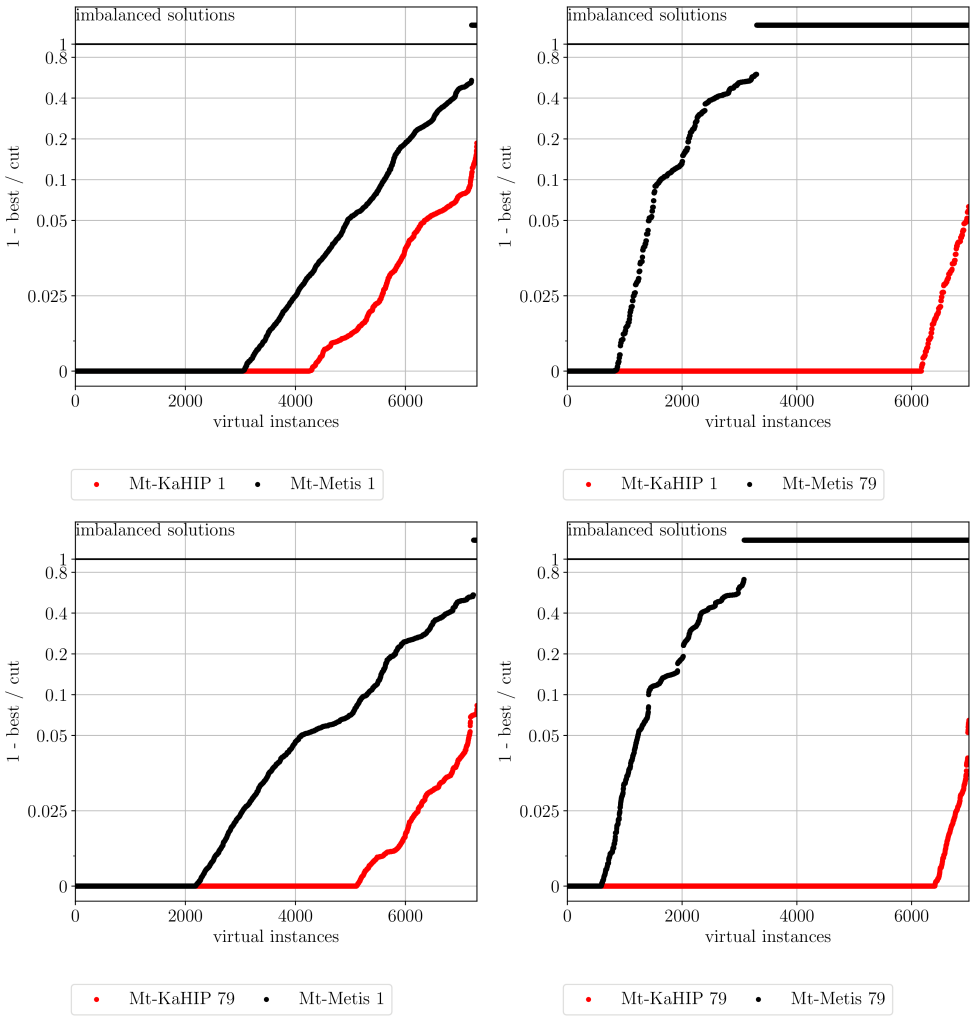


Figure 4.8: Effectiveness tests for `Mt-KaHIP` and `Mt-Metis`. The number behind the algorithm name denotes the number of PEs used.

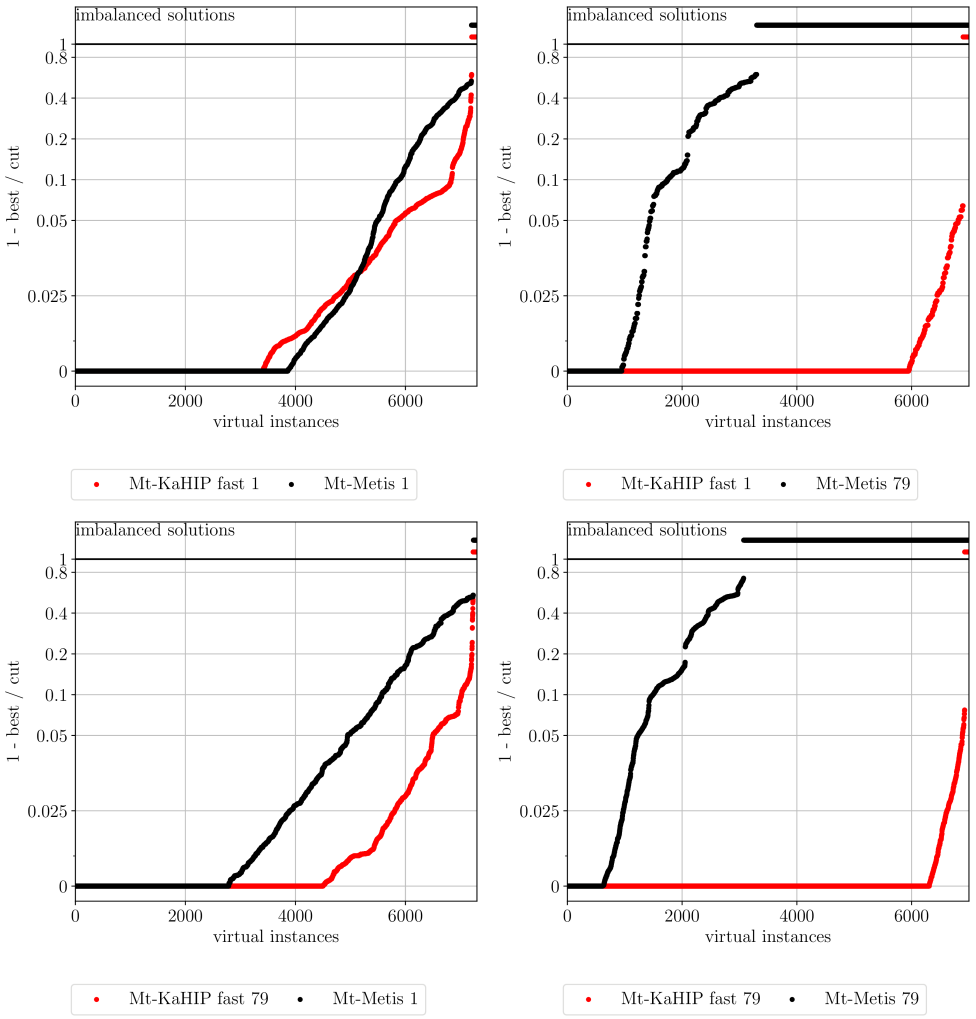


Figure 4.9: Effectiveness tests for `Mt-KaHIP fast` and `Mt-Metis`. The number behind the algorithm name denotes the number of PEs used.

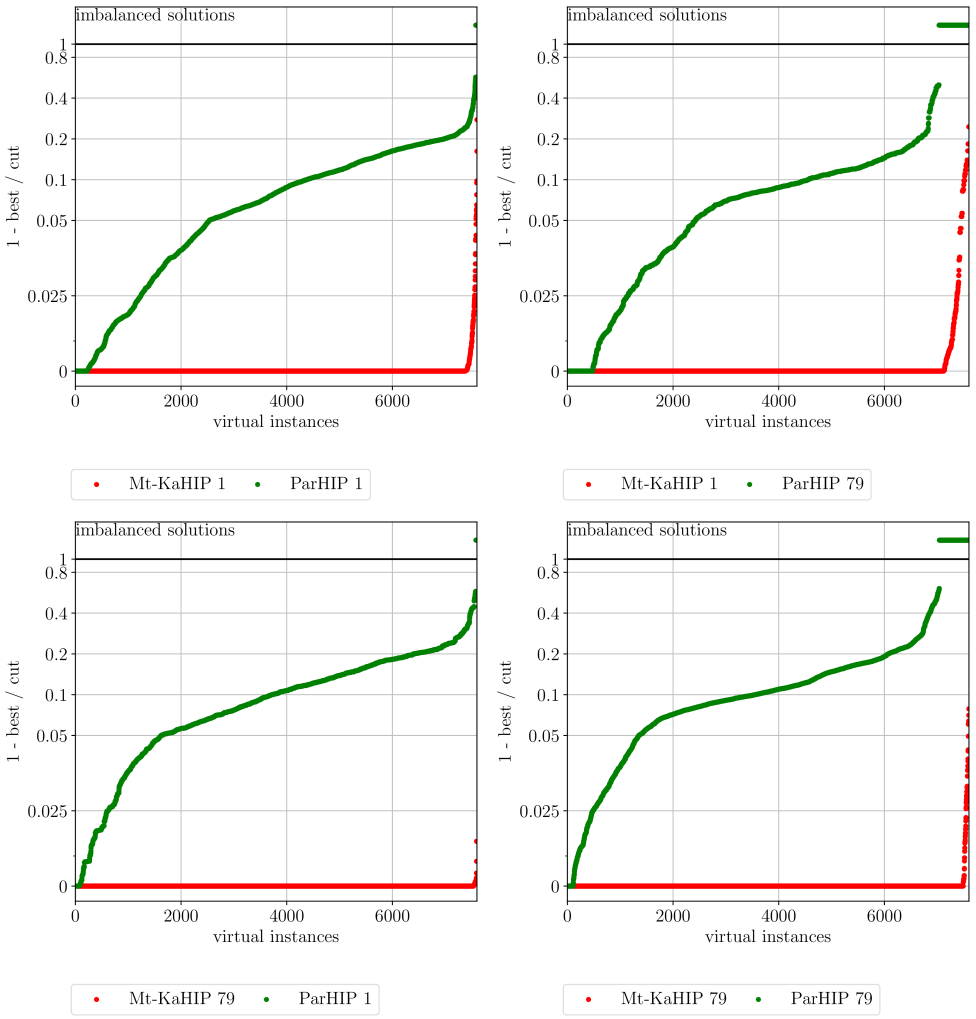


Figure 4.10: Effectiveness tests for `Mt-KaHIP` and `ParHIP`. The number behind the algorithm name denotes the number of PEs used.

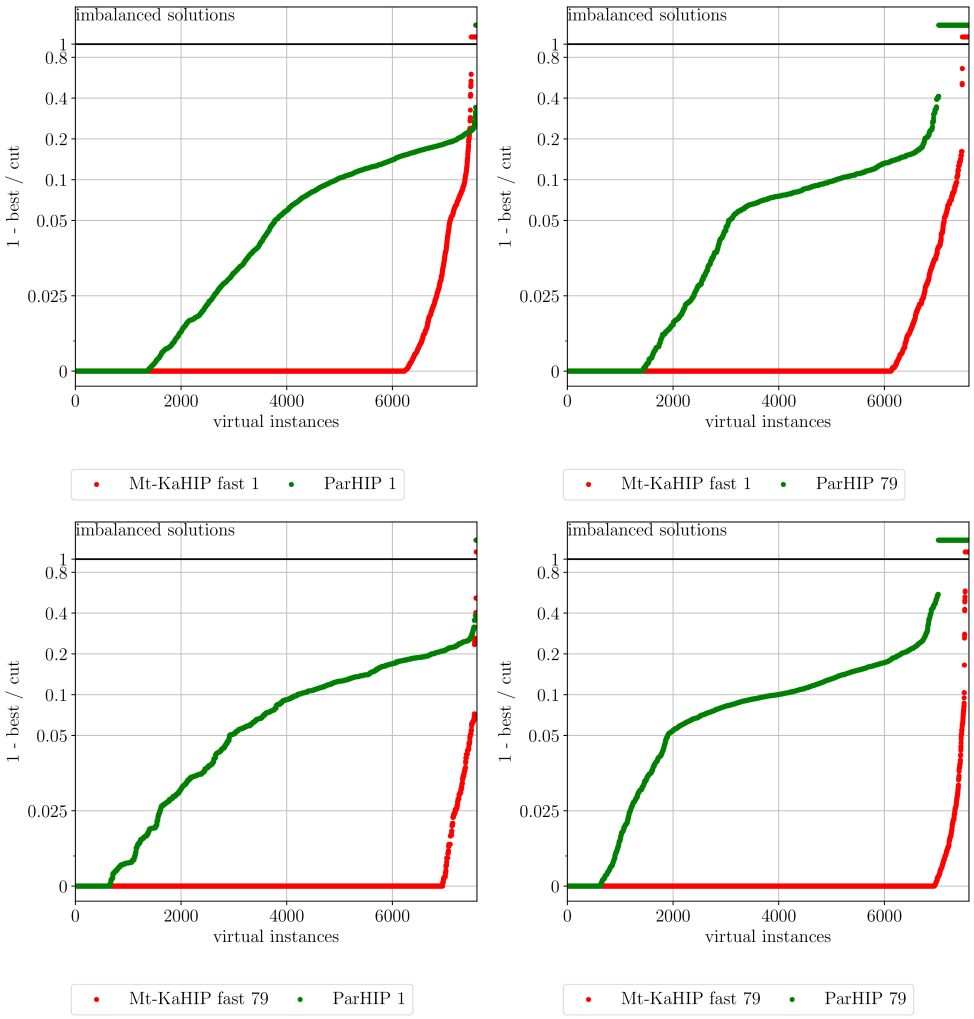


Figure 4.11: Effectiveness tests for `Mt-KaHIP fast` and `ParHIP`. The number behind the algorithm name denotes the number of PEs used.

Additional Quality Comparisons

Here we present additional quality comparisons of `Mt-KaHIP` to `Mt-KaHIP eco`, `DiBaP`, and `PuLP`.

Comparison to `Mt-KaHIP eco`. We additionally compare `Mt-KaHIP` to `Mt-KaHIP eco`, which uses the local max matching algorithm during coarsening, on all 16 mesh type graphs (32 instances) from the set of instances $\mathcal{I}$ except the graph rhg. We consider `Mt-KaHIP eco` on the mesh type graphs since they do not have communities which the label propagation algorithm searches for during the coarsening phase of `Mt-KaHIP`. Note that matching algorithms are not biased towards communities. Therefore, we investigate how the parallel local max matching algorithm during the coarsening phase affects the resulting quality of our framework.

Figure 4.12 shows a performance plot (see Section 2.3.1) that compares `Mt-KaHIP` and `Mt-KaHIP eco`. We use the symmetric log scaling on the y-axis which allows to define a range around zero that is scaled linearly, whereas the rest of the y-axis is scaled logarithmically. In this performance plot, we have a linear scaling up to 0.05 since most of the points are concentrated below 0.05. `Mt-KaHIP` and `Mt-KaHIP eco` produce the best partition on 14 and 18 instances, respectively. `Mt-KaHIP eco` has a 5.1% smaller cut than `Mt-KaHIP` in the best case and a 12.5% larger cut than `Mt-KaHIP` in the worst case. The geometric mean cut sizes of `Mt-KaHIP` and `Mt-KaHIP eco` are $1390.3K$ and $1394.8K$, respectively. The Wilcoxon signed-rank test (see Section 2.3.3) indicates that the quality advantage of `Mt-KaHIP` over `Mt-KaHIP eco` is not statistically significant. The p-value is 0.30. Furthermore, note that `Mt-KaHIP eco` runs longer on almost all instances. See details in Section 4.5.3.

Figure 4.13 shows a performance plot (see Section 2.3.1) that compares `Mt-KaHIP`, `Mt-KaHIP eco`, and `Mt-Metis` with $p = 1, 79$. We additionally compare ourselves to `Mt-Metis` with $p = 1$ to show that `Mt-KaHIP` has comparable quality to `Mt-Metis` with $p = 1$ which in turn has better quality than `Mt-Metis` with 79 PEs. Furthermore, `Mt-Metis` with a single PE almost always produces balanced partitions unlike `Mt-Metis` with 79 PEs. `Mt-KaHIP`, `Mt-KaHIP eco`, and `Mt-Metis` with $p = 1$ produce the best partition on 9, 11, and 12 instances, respectively. `Mt-KaHIP`, `Mt-KaHIP eco`, and `Mt-Metis` with $p = 1, 79$ have a 7.8%, 14.7%, 9.9%, and 18.5% larger cut than the best cut in the worst case. The geometric mean cut sizes of `Mt-Metis` with $p = 1$ is $1389.2K$. The geometric mean cut sizes of `Mt-Metis` with $p = 79$ is $1391.0K$ (note that `Mt-Metis` with $p = 79$ was not able to partition one instance). The Wilcoxon signed-rank test indicates that the quality advantage of `Mt-Metis` over `Mt-KaHIP` and `Mt-KaHIP eco` is not statistically significant. The p-values are 0.12 and 0.61, respectively.

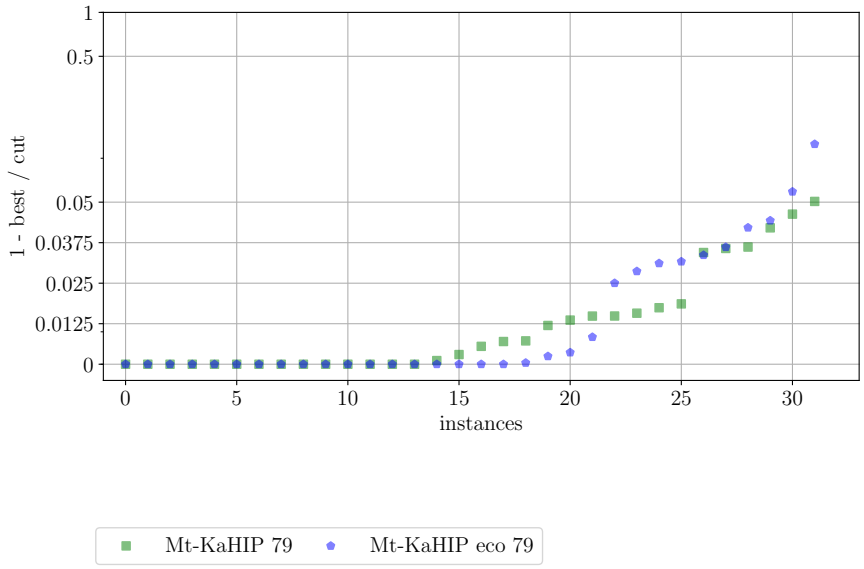


Figure 4.12: Performance plot for the cut size of `Mt-KaHIP` and `Mt-KaHIP eco`. The number behind the algorithm name denotes the number of PEs used.

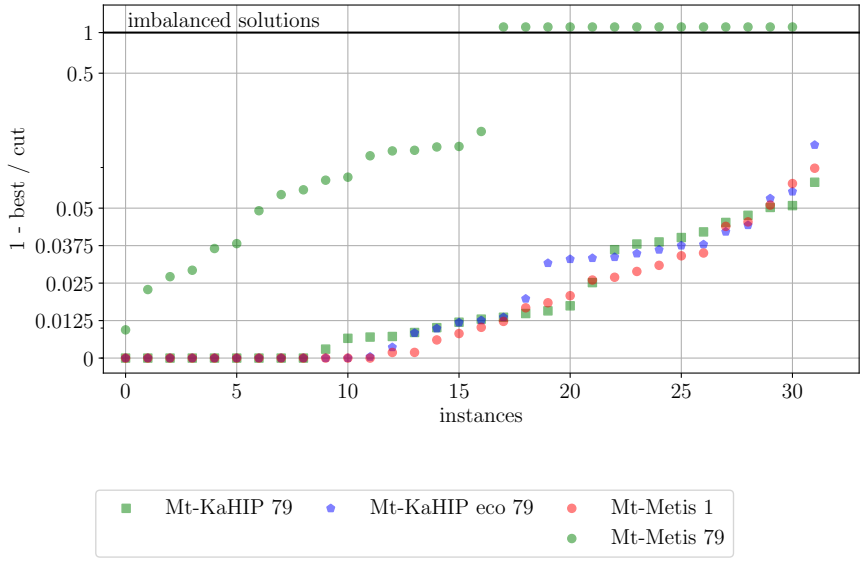


Figure 4.13: Performance plot for the cut size of `Mt-KaHIP`, `Mt-KaHIP eco`, and `Mt-Metis`.

Comparison to DiBaP. We present a quality comparison of Mt-KaHIP and DiBaP which produces partitions of good quality. Unfortunately, we were not able to run DiBaP on our main benchmark. Therefore, we were advised by the authors of DiBaP to compare the frameworks using the cut sizes by Meyerhenke et al. [MMS09a]. We run our framework on machine A. Table 4.7 presents the cut sizes of partitions produced by Mt-KaHIP and parallel DiBaP. We additionally present the results of kMetis since they are used as a baseline comparison in the original paper [MMS09a]. The kMetis columns contain the average cut size for different number of blocks. Then for each framework we compute cut size ratios (cut size of the framework divided by kMetis) for each graph and seed and present the average ratio. We consider the default configuration of Mt-KaHIP and its strong configuration where LMLS is allowed to spend more time to find better solution (we compute 99.99999% quantiles in the stopping rules of LMLS). We select quantiles such that the running time ratio between Mt-KaHIP default and Mt-KaHIP strong is approximately equal to that of between DiBaP-long and DiBaP-short. The reason behind it is that we cannot compare absolute running times of the frameworks since they were evaluated on different machines. Table 4.8 shows ratios between Mt-KaHIP default and Mt-KaHIP strong as well as speed-ups of the frameworks. Furthermore, we evaluate Mt-KaHIP for $p = 1$ and $p = 2$ since DiBaP was also evaluated for the same numbers of PEs. Meyerhenke et al. [MMS09a] state that they have a speed-up of 1.55 and quality does not change for $p = 2$. Mt-KaHIP default shows comparable quality to that of DiBaP-short and a speed-up of 1.58, whilst Mt-KaHIP strong shows better quality to that of DiBaP-long and a speed-up of 1.68.

Table 4.7: Cut sizes of `kMetis`, `Mt-KaHIP`, and `DiBaP`.

Blocks	`kMetis`	`Mt-KaHIP` default		`Mt-KaHIP` strong		`DiBaP` short	`DiBaP` long
		$p=1$	$p=2$	$p=1$	$p=2$		
4	13836.9	0.917	0.917	**0.898**	0.905	0.971	0.926
8	24079.0	0.939	0.933	0.923	**0.922**	0.945	0.952
16	39013.7	0.945	0.944	0.928	**0.922**	0.927	0.924
32	58275.7	0.959	0.963	0.934	0.935	**0.909**	0.939
64	82292.8	0.974	0.977	0.947	**0.945**	0.980	0.948
Harmonic mean (rel.)	1.000	0.947	0.946	0.926	**0.926**	0.946	0.938

Table 4.8: Relative running times of `Mt-KaHIP`, and `DiBaP`.

Blocks	`Mt-KaHIP` ($\frac{\text{strong}}{\text{default}}$)		`DiBaP` ($\frac{\text{strong}}{\text{default}}$)	`Mt-KaHIP` speed-up		`DiBaP` speed-up
	$p=1$	$p=2$		default	strong	
4	1.24	**1.17**	2.27	1.54	**1.64**	1.55
8	1.45	**1.39**	2.38	1.57	**1.63**	1.55
16	1.89	**1.81**	2.45	1.61	**1.68**	1.55
32	2.39	**2.22**	2.64	1.62	**1.75**	1.55
64	3.19	2.92	**2.72**	1.58	**1.72**	1.55
Harmonic mean (rel.)	1.82	1.71	2.48	1.58	1.68	1.55

Comparison to PuLP. We present a quality comparison of Mt-KaHIP, Mt-KaHIP fast and PuLP. The experiments were performed on machine A. The performance plots (see Section 2.3.1) in Figures 4.14 and 4.15 show the results of our experiments. We use the symmetric log scaling on the y-axis which allows to define a range around zero that is scaled linearly, whereas the rest of the y-axis is scaled logarithmically. In this performance plot, we have a linear scaling up to 0.05 since most of the points are concentrated below 0.05. Our algorithm gives the best overall quality, usually producing the best cut for most instances. There are only two instances which PuLP partitions better and even in these cases Mt-KaHIP with $p = 79$ is at most 2.0% worse. Table 4.9 shows the geometric mean cut sizes of the frameworks over the set of instances $\mathcal{S}_{\texttt{PuLP}}$. We can see that the geometric mean cut sizes of Mt-KaHIP and Mt-KaHIP fast are about two times smaller than that of PuLP. Furthermore, the Wilcoxon signed-rank test (see Section 2.3.3) indicates that the quality advantage of Mt-KaHIP and Mt-KaHIP fast with $p = 79$ over PuLP with $p = 79$ is statistically significant. Both p-values are $2 \cdot 10^{-12}$.

Table 4.9: Geometrical means of cut sizes of Mt-KaHIP, Mt-KaHIP fast, and PuLP over the set of instances $\mathcal{S}_{\texttt{PuLP}}$.

	Number of processors		
Algorithm	1	40	79
Mt-KaHIP	**2000.8*K***	**1975.4*K***	**1973.2*K***
Mt-KaHIP fast	2095.4K	2081.3K	2075.9K
Mt-Metis	5100.0K	5208.7K	5474.4K

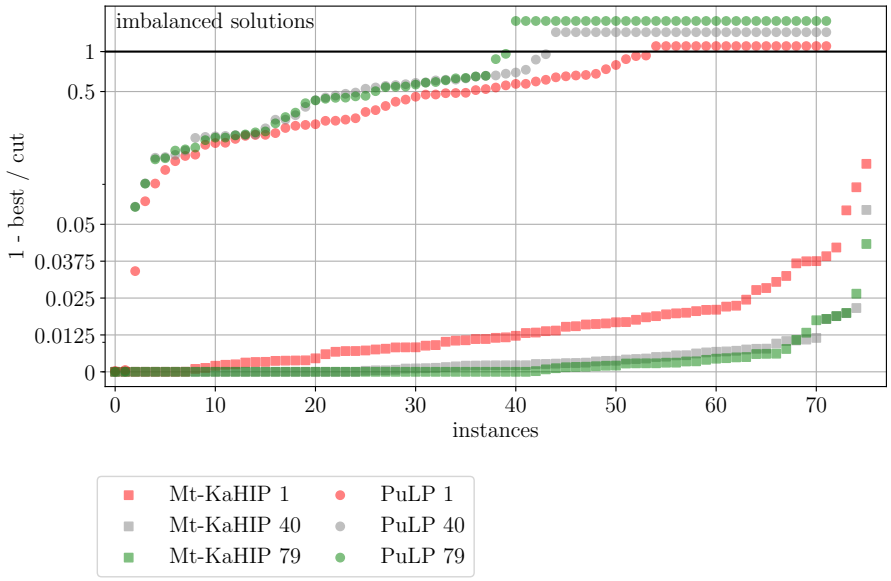


Figure 4.14: Performance plot for the cut size of `Mt-KaHIP` and `PuLP`. The number behind the algorithm name denotes the number of PEs used.

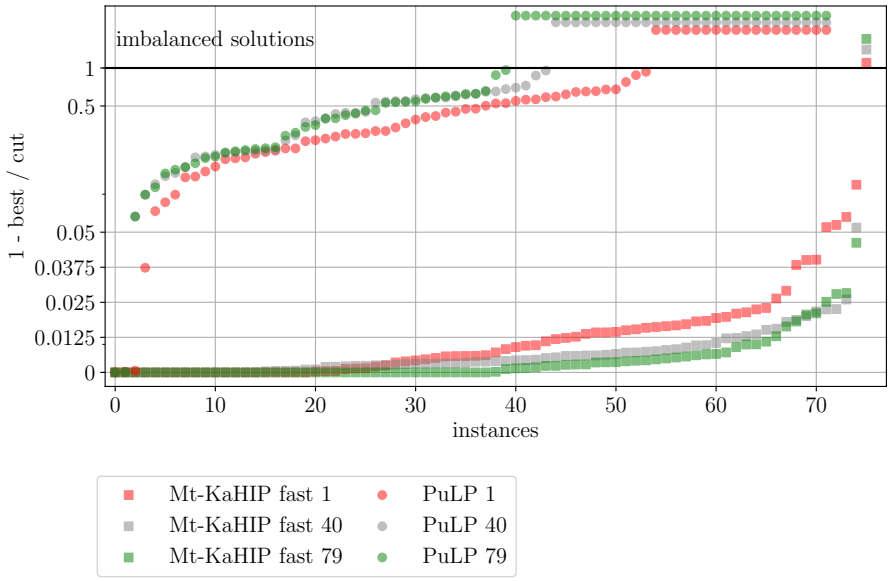


Figure 4.15: Performance plot for the cut size of `Mt-KaHIP fast` and `PuLP`.

Additionally, Figure 4.16 presents the effectiveness tests that compare `Mt-KaHIP` and `Mt-KaHIP fast` to `PuLP` for $p = 79$. We use the symmetric log scaling on the y-axis which allows to define a range around zero that is scaled linearly, whereas the rest of the y-axis is scaled logarithmically. In this performance plot, we have a linear scaling up to 0.05 since most of the points are concentrated below 0.05. Specifically, `Mt-KaHIP` has better quality than `PuLP` for 97.2% of the virtual instances and their geometric mean cut sizes are 1972.7 and 4509.3, respectively. The Wilcoxon signed-rank test (see Section 2.3.3) indicates that the quality advantage of `Mt-KaHIP` with $p = 79$ over `PuLP` with $p = 79$ is statistically significant. The p-values is 0. Furthermore, `Mt-KaHIP` has a 99.4% smaller cut than `PuLP` in the best case and a 2.3% larger cut than `PuLP` in the worst case. `Mt-KaHIP fast` has better quality than `PuLP` for 96.9% of the virtual instances and their geometric mean cut sizes are 2071.2 and 4631.5, respectively. The Wilcoxon signed-rank test (see Section 2.3.3) indicates that the quality advantage of `Mt-KaHIP fast` and `PuLP` with $p = 79$ is statistically significant. The p-values is 0. Furthermore, `Mt-KaHIP fast` has a 99.4% smaller cut than `PuLP` in the best case and a 51.8% larger cut than `PuLP` in the worst case. However, if we consider only instances partitioned without imbalance then `Mt-KaHIP fast` has 3.9% larger cut than `PuLP` in the worst case and 98.9% smaller cut in the best case.

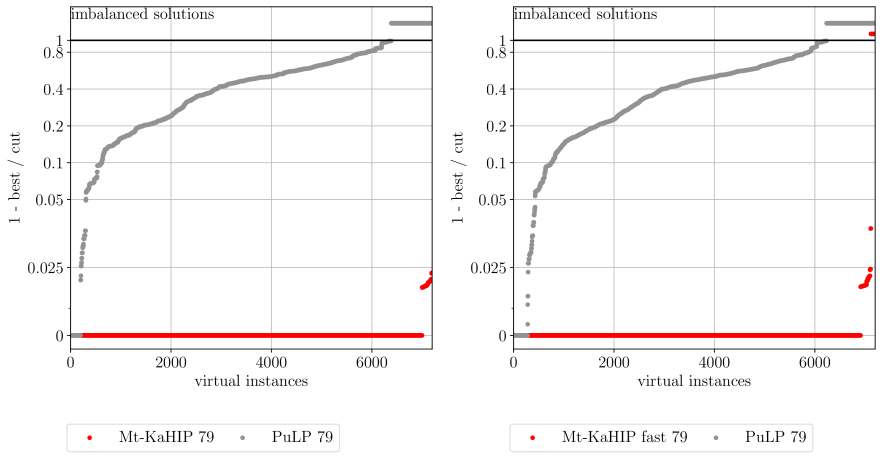


Figure 4.16: Effectiveness tests for `Mt-KaHIP`, `Mt-KaHIP fast` and `PuLP`. The number behind the algorithm name denotes the number of PEs used.

4.5.3 Speed-up and Running Time

In this section, we analyze speed-ups and running times of `Mt-KaHIP`, `Mt-KaHIP fast`, `Mt-Metis` and `ParHIP`. Furthermore, we analyze performance of their main components: the coarsening phases, the uncoarsening phases, and local searches. Here a relative speed-up of an algorithm with p PEs on an instance is the ratio between its running time with a single PE (averaged over ten repetitions) and its running time with p PEs (averaged over ten repetitions). We compare LMLS from `Mt-KaHIP` and the hill-climbing technique from `Mt-Metis` as well as the label propagation implementations of `Mt-KaHIP` and `ParHIP`. Additionally, we compare ourselves to `PuLP`. We performed experiments on machines A and B to investigate how NUMA effects affect performance.

Complete algorithms

In this paragraph, we compare performance of complete runs of `Mt-KaHIP`, `Mt-KaHIP fast`, `Mt-Metis`, and `ParHIP` on machines A and B.

Machine A. Here we present an analysis of complete runs of the frameworks. Figure 4.17 shows scatter plots with the speed-ups and running times per edge. We can see that the speed-ups and times per edge of `Mt-KaHIP` improve with increasing number of edges. Note that with increasing number of edges, `Mt-KaHIP` has better speed-ups than `Mt-Metis` and better times per edge than those of `ParHIP`. Moreover, 85% of the twenty highest time per edge relative differences ($|a-b|/\max(a,b)$) between `Mt-Metis` and `Mt-KaHIP` – where `Mt-Metis` is faster – correspond to the instances partitioned with imbalance (Mt-Metis 79 imbalanced). `Mt-KaHIP fast` has better speed-ups than `Mt-Metis` and `ParHIP`. Furthermore, `Mt-KaHIP fast` has comparable times per edge to those of `Mt-Metis` and better times per edge than `ParHIP`. To analyze the overall performance, we consider cumulative harmonic mean speed-ups and geometric mean running times presented on Figures 4.18 and 4.19.

Table 4.10 shows the harmonic mean speed-ups and the geometric mean running times. `Mt-KaHIP` has slightly worse harmonic mean speed-up and geometric mean running time than other frameworks since the coarsest graph is not small enough and the initial partitioning phase becomes a bottleneck. See Section 4.5.5 for details. However, note that `Mt-KaHIP fast` does not have this problem and has better harmonic mean speed-up than other frameworks and better geometric mean running time than `ParHIP`. Although `Mt-KaHIP` has worse harmonic mean speed-up than other competitors on the full set of instances $\mathcal{S}_{\texttt{Mt-Metis}}$, Figure 4.18 shows that if we exclude the graphs with less than 43M edges then the harmonic mean speed-up of `Mt-KaHIP` is better than that of `Mt-Metis`. Furthermore, Figures 4.18 and 4.19 show that excluding graphs with less than about 150M edges improves the geometric mean running time of `Mt-KaHIP` such that it is better than that of `ParHIP`. In summary, although on

Table 4.10: Harmonic mean speed-ups and geometric mean running times of the frameworks over different sets of instances. Here $\mathcal{S}_{\texttt{Mt-Metis}}$ *balanced* is 22 instances from $\mathcal{S}_{\texttt{Mt-Metis}}$ that are partitioned without imbalance by all frameworks.

	Speed-up			Running time (s)		
Algorithm	$\mathcal{S}_{\texttt{Mt-Metis}}$	$\mathcal{S}_{\texttt{Mt-Metis}}$ *balanced*	$\mathcal{I}$	$\mathcal{S}_{\texttt{Mt-Metis}}$	$\mathcal{S}_{\texttt{Mt-Metis}}$ *balanced*	$\mathcal{I}$
`Mt-KaHIP`	6.1	19.1	5.9	18.8	15.7	21.9
`Mt-KaHIP fast`	**12.4**	25.5	**12.8**	8.5	12.5	**9.2**
`Mt-Metis`	8.5	10.9	–	**5.7**	**6.0**	–
`ParHIP`	7.4	**31.1**	7.3	15.8	15.5	18.1

average our algorithm is slower than `Mt-Metis`, our framework has better quality (see Section 4.5.2) and we consider this as a fair trade off between the quality and the running time. Furthermore, we dominate `ParHIP` in terms of quality and running times if we exclude small graphs.

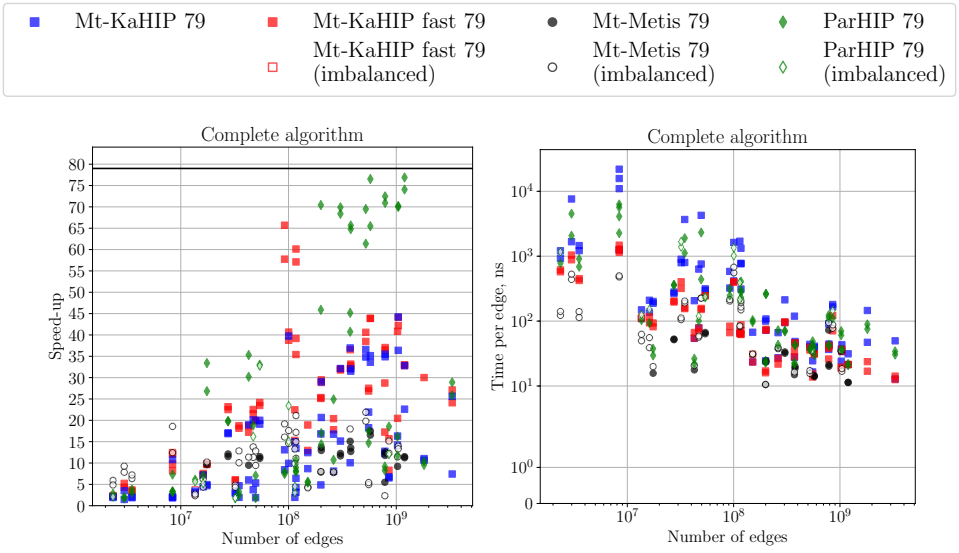


Figure 4.17: Scatter plots with speed-ups and running times per edge of the frameworks for $p = 79$.

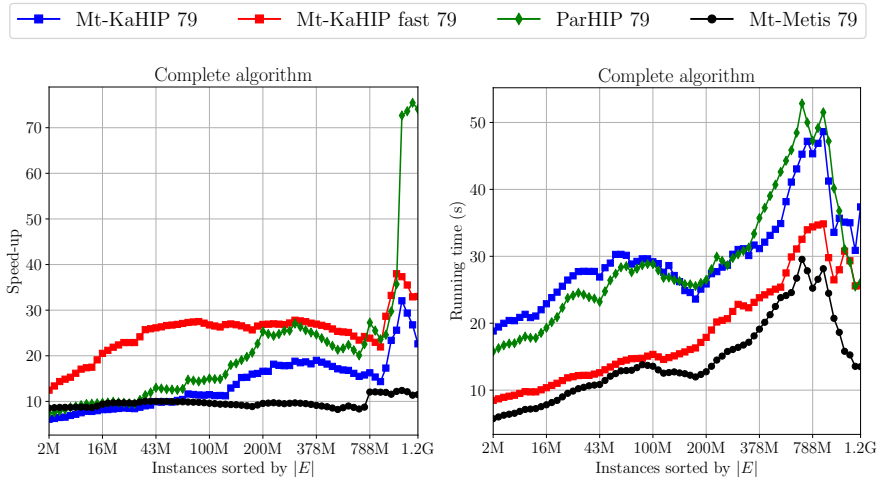


Figure 4.18: *Cumulative* harmonic mean speed-ups and geometric mean running times of the frameworks for $p = 79$ on the set of instances $\mathcal{S}_{\texttt{Mt-Metis}}$. Here a point (x, y) means that the harmonic mean speed-up (geometric mean running time) of the graphs with $|E| \geq x$ is y.

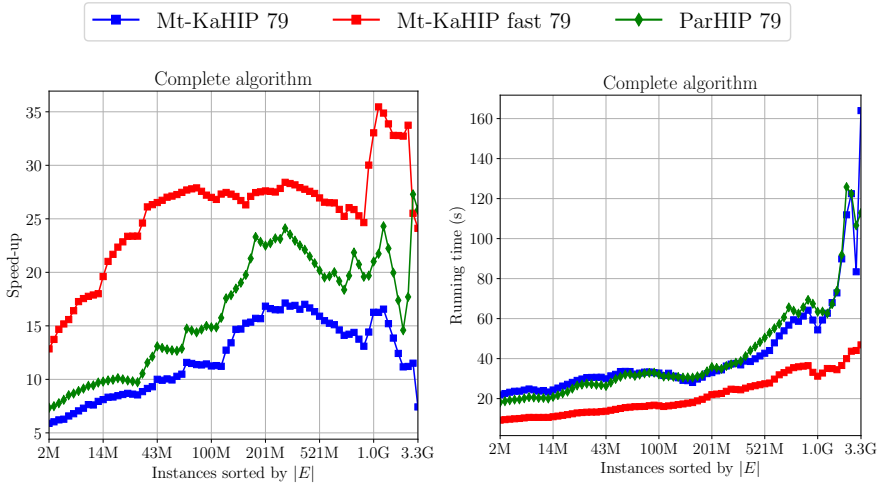


Figure 4.19: *Cumulative* harmonic mean speed-ups and geometric mean running times of `Mt-KaHIP` and `ParHIP` for $p = 79$ on the set of instances $\mathcal{I}$.

Machine B. Here we present an analysis of complete runs of the frameworks. Figure 4.20 shows the scatter plots with speed-ups and running times per edge of the frameworks. We can see that speed-ups and times per edge of `Mt-KaHIP` improve with increasing number of edges. Note that with increasing number of edges, `Mt-KaHIP` has better speed-ups and times per edge to those of `ParHIP` as well as better speed-ups than `Mt-Metis`. Moreover, 90% of the twenty highest time per edge relative differences ($|a - b|/\max(a, b)$) between `Mt-Metis` and `Mt-KaHIP` – where `Mt-Metis` is faster – correspond to the instances partitioned with imbalance (Mt-Metis 31 imbalanced). Furthermore, to analyze the overall performance of the frameworks, we consider harmonic mean speed-ups and geometric mean running times presented on Figures 4.21.

Table 4.11 shows the harmonic mean speed-ups and the geometric mean running times of the frameworks. `Mt-KaHIP` has slightly worse harmonic mean speed-up than `Mt-Metis` since the coarsest graph is not small enough and the initial partitioning phase becomes a bottleneck. However, Figure 4.21 shows that if we exclude the graphs with the number of edges less than 47M then the harmonic mean speed-up of `Mt-KaHIP` is better than that of `Mt-Metis`. In summary, although on average our algorithm is slower than `Mt-Metis`, we consider this as a fair trade off between the quality and the running time. Furthermore, we dominate `ParHIP` in terms of quality and running times.

Table 4.11: Harmonic mean speed-ups and geometric mean running times of the frameworks over different sets of instances. Here $\mathcal{S}^B_{\texttt{Mt-Metis}}$ *balanced* is 22 instances from $\mathcal{S}^B_{\texttt{Mt-Metis}}$ that are partitioned without imbalance.

	Instances							
	Speed-up				Running time (s)			
Algorithm	$\mathcal{S}^B_{\texttt{Mt-Metis}}$	$\mathcal{S}^B_{\texttt{Mt-Metis}}$ *balanced*	$\mathcal{S}^B_{\texttt{ParHIP}}$	$\mathcal{S}^B_{\texttt{All}}$	$\mathcal{S}^B_{\texttt{Mt-Metis}}$	$\mathcal{S}^B_{\texttt{Mt-Metis}}$ *balanced*	$\mathcal{S}^B_{\texttt{ParHIP}}$	$\mathcal{S}^B_{\texttt{All}}$
`Mt-KaHIP`	8.2	17.9	8.1	8.5	20.3	17.8	23.7	20.1
`Mt-Metis`	10.2	11.1	–	10.5	6.9	7.0	36.4	7.0
`ParHIP`	–	–	6.4	6.4	–	–	–	32.0

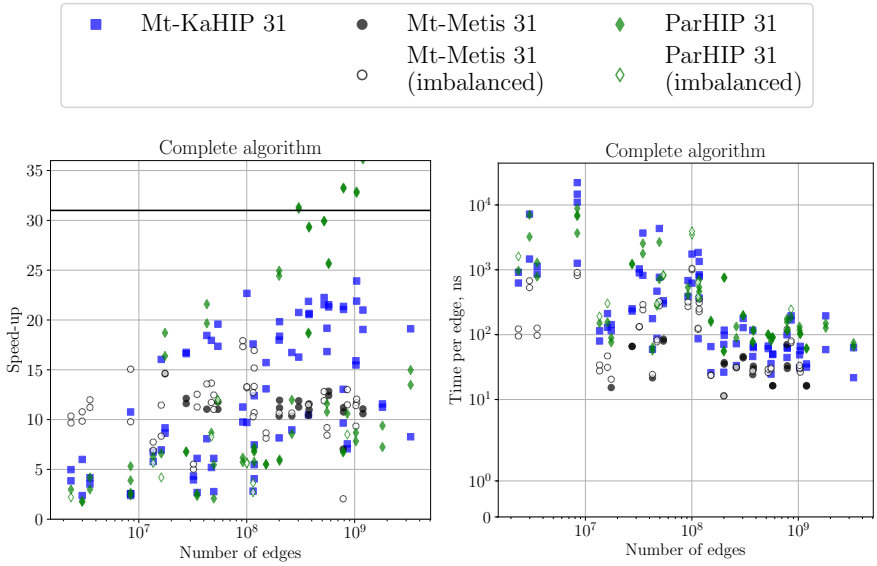


Figure 4.20: Scatter plots with speed-ups and average running times per edge of the frameworks for $p = 31$.

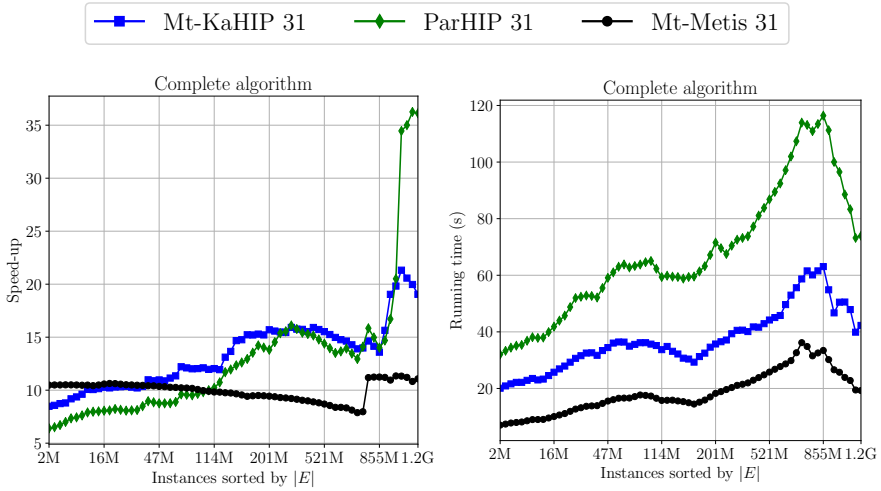


Figure 4.21: *Cumulative* harmonic mean speed-ups and geometric mean running times of the frameworks for $p = 31$ on the set of instances $\mathcal{S}^{B}_{\texttt{All}}$. Here a point (x, y) means that the harmonic mean speed-up (geometric mean running time) of the graphs with $|E| \geq x$ is y.

Comparison of Components

Here we compare performance of the MGP phases of the frameworks and local search techniques used in them on machine A. Performance of the components on machine B is similar.

Figure 4.22 shows the scatter plots with speed-ups and running times per edge of the coarsening and uncoarsening phases as well as the combination of them. Furthermore, Figure 4.22 shows comparison of local searches. Specifically, we compare parallel LMLS used in `Mt-KaHIP` against the hill-climbing technique used in `Mt-Metis` and the parallel label propagation algorithms used in `Mt-KaHIP` and `ParHIP`.

We can see that the speed-ups and times per edge of `Mt-KaHIP` improve with increasing number of edges for all components. Note that with increasing number of edges, the components of `Mt-KaHIP` have better speed-ups and times per edge than those of `Mt-Metis` and of `ParHIP`. Furthermore, to analyze the overall performance of the components, we compare harmonic mean speed-ups and geometric mean running times of `Mt-KaHIP`, `Mt-Metis`, and `ParHIP` in pairs.

Figure 4.23 shows the comparison of components of `Mt-KaHIP` and `Mt-Metis`. Additionally, Table 4.12 shows the harmonic mean speed-up and geometric mean running time of `Mt-KaHIP` and `Mt-Metis` on the set of instances $\mathcal{S}_{\texttt{Mt-Metis}}$. We observe that `Mt-KaHIP` has better harmonic mean speed-ups of the components than those of `Mt-Metis`. Furthermore, if we exclude the graphs with less than about 250M edges then the uncoarsening phase of `Mt-KaHIP` has smaller geometric mean running time than that of `Mt-Metis`. If we exclude the graphs with less than about 43M edges then the parallelized LMLS of `Mt-KaHIP` has smaller geometric mean running time than the hill-climbing technique of `Mt-Metis`. The coarsening phase of `Mt-KaHIP` has larger geometric mean running times than that of `Mt-Metis` since we use the parallel label propagation algorithm in the coarsening phase which performs more work than the parallel heavy-edge matching algorithm used in `Mt-Metis`.

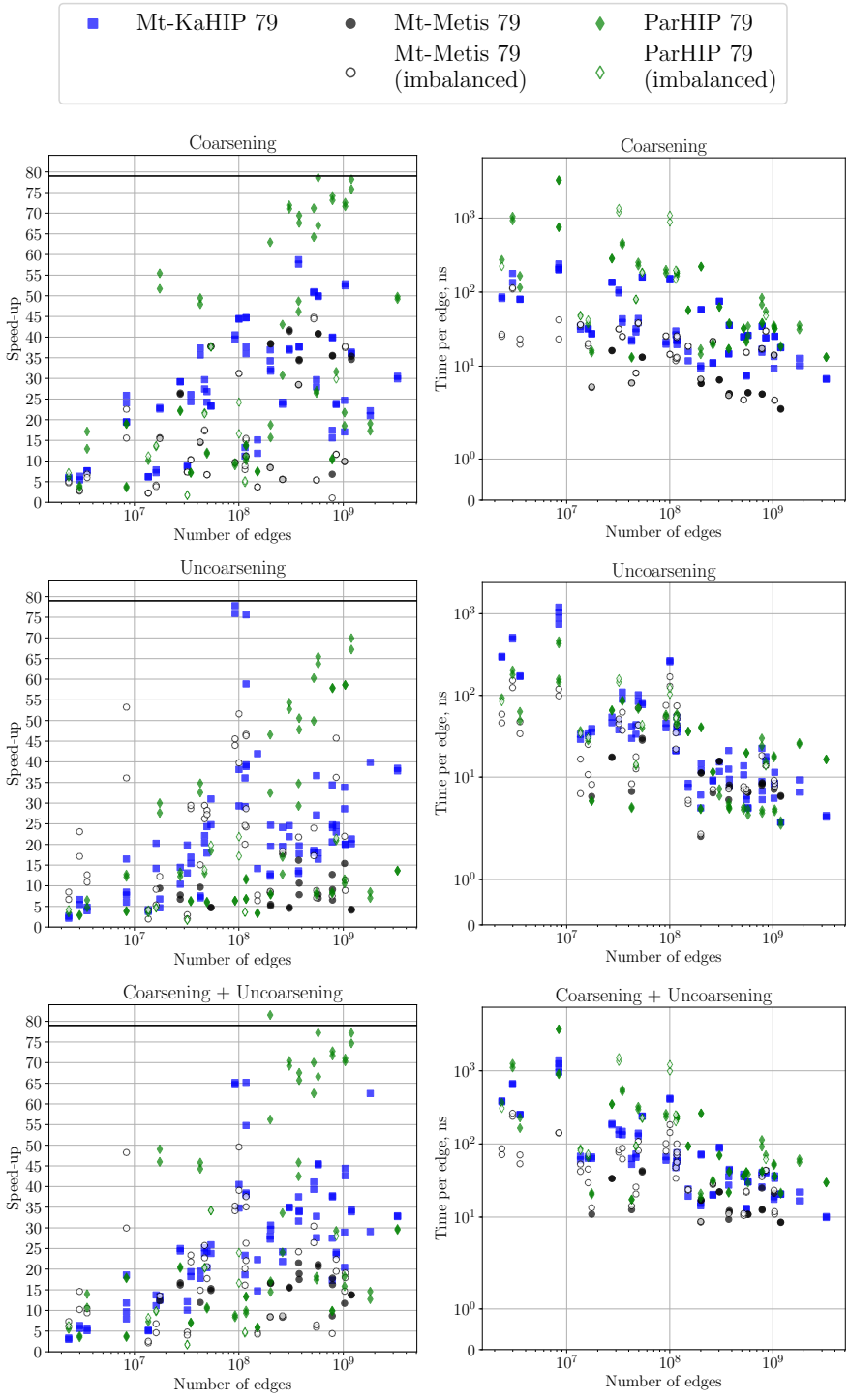
Figure 4.24 shows the comparison of `Mt-KaHIP` and `ParHIP`. Additionally, Table 4.13 shows the harmonic mean speed-up and geometric mean running time of `Mt-KaHIP` and `ParHIP` on the set of instances $\mathcal{I}$. We observe that `Mt-KaHIP` has better harmonic mean speed-ups of the components than those of `ParHIP` except the harmonic speed-up of the parallel label propagation algorithm. However, if we exclude the smallest graph with 2M edges then the parallel label propagation algorithm of `Mt-KaHIP` has better harmonic mean speed-up than that of `ParHIP`. `Mt-KaHIP` has better geometric mean running times of the components than those of `ParHIP` except the geometric mean running time of the uncoarsening phase. However, if we consider only graphs that have more than 43M edges then the uncoarsening phase of `Mt-KaHIP` has better geometric mean running times. This is not surprising since the uncoarsening phase in `Mt-KaHIP` consists of the parallel label propagation and LMLS algorithms, whereas it consists of only the parallel label propagation algorithm in `ParHIP`.

Table 4.12: Comparison of the components of `Mt-KaHIP` and `Mt-Metis` on the set of instances $\mathcal{S}_{\texttt{Mt-Metis}}$. Each cell of the table contains the harmonic mean speed-up and the geometric mean running time. Bold numbers are the best speed-ups and running times for components.

	Coarsening		Uncoarsening		Coarsening + Uncoarsening		Local search	
With imbalanced instances								
`Mt-KaHIP`	**18.3**	3.5 s	**12.4**	2.9 s	**16.1**	7.0 s	**13.1**	1.8 s
`Mt-Metis`	8.2	**1.3** s	9.0	**1.7** s	10.9	**3.3** s	8.1	**1.5** s
Without imbalanced instances								
`Mt-KaHIP`	**30.5**	8.1 s	**15.8**	2.8 s	**27.2**	11.9 s	**11.2**	**1.4** s
`Mt-Metis`	19.8	**1.6** s	6.9	**2.1** s	14.2	**3.9** s	6.0	1.6 s

Table 4.13: Comparison of the components of `Mt-KaHIP` and `ParHIP` on the set of instances $\mathcal{I}$. Each cell of the table contains the harmonic mean speed-up and the geometric mean running time. Bold numbers are the best speed-ups and running times for components.

	Coarsening		Uncoarsening		Coarsening + Uncoarsening		Local search	
With imbalanced instances								
`Mt-KaHIP`	**18.6**	**3.8** s	**12.8**	3.2 s	**16.5**	**7.7** s	7.7	**0.6** s
`ParHIP`	12.2	9.9 s	8.0	**2.6** s	11.2	13.0 s	**8.0**	2.6 s
Without imbalanced instances								
`Mt-KaHIP`	**20.9**	**4.0** s	**13.1**	3.2 s	**17.9**	**8.1** s	8.4	**0.6** s
`ParHIP`	15.5	9.7 s	9.4	**2.6** s	13.9	12.9 s	**9.4**	2.6 s


Mt-KaHIP 79
Mt-Metis 79
Mt-Metis 79 (imbalanced)
ParHIP 79
ParHIP 79 (imbalanced)
Coarsening
Speed-up
Number of edges
Coarsening
Time per edge, ns
Number of edges
Uncoarsening
Speed-up
Number of edges
Uncoarsening
Time per edge, ns
Number of edges
Coarsening + Uncoarsening
Speed-up
Number of edges
Coarsening + Uncoarsening
Time per edge, ns
Number of edges

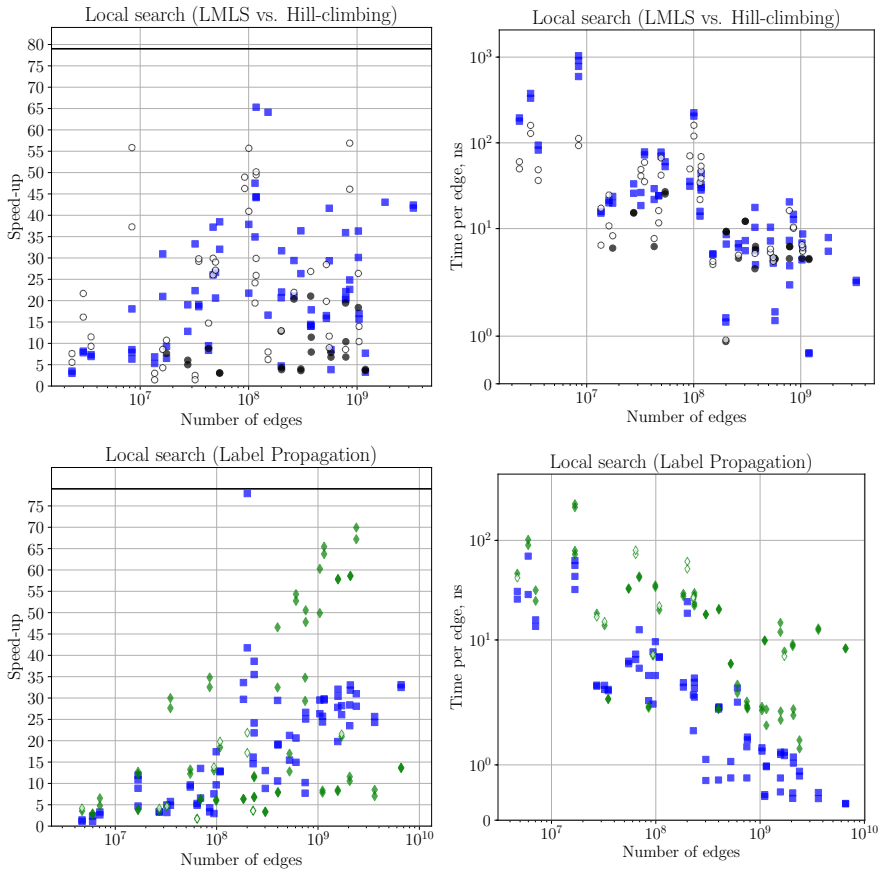
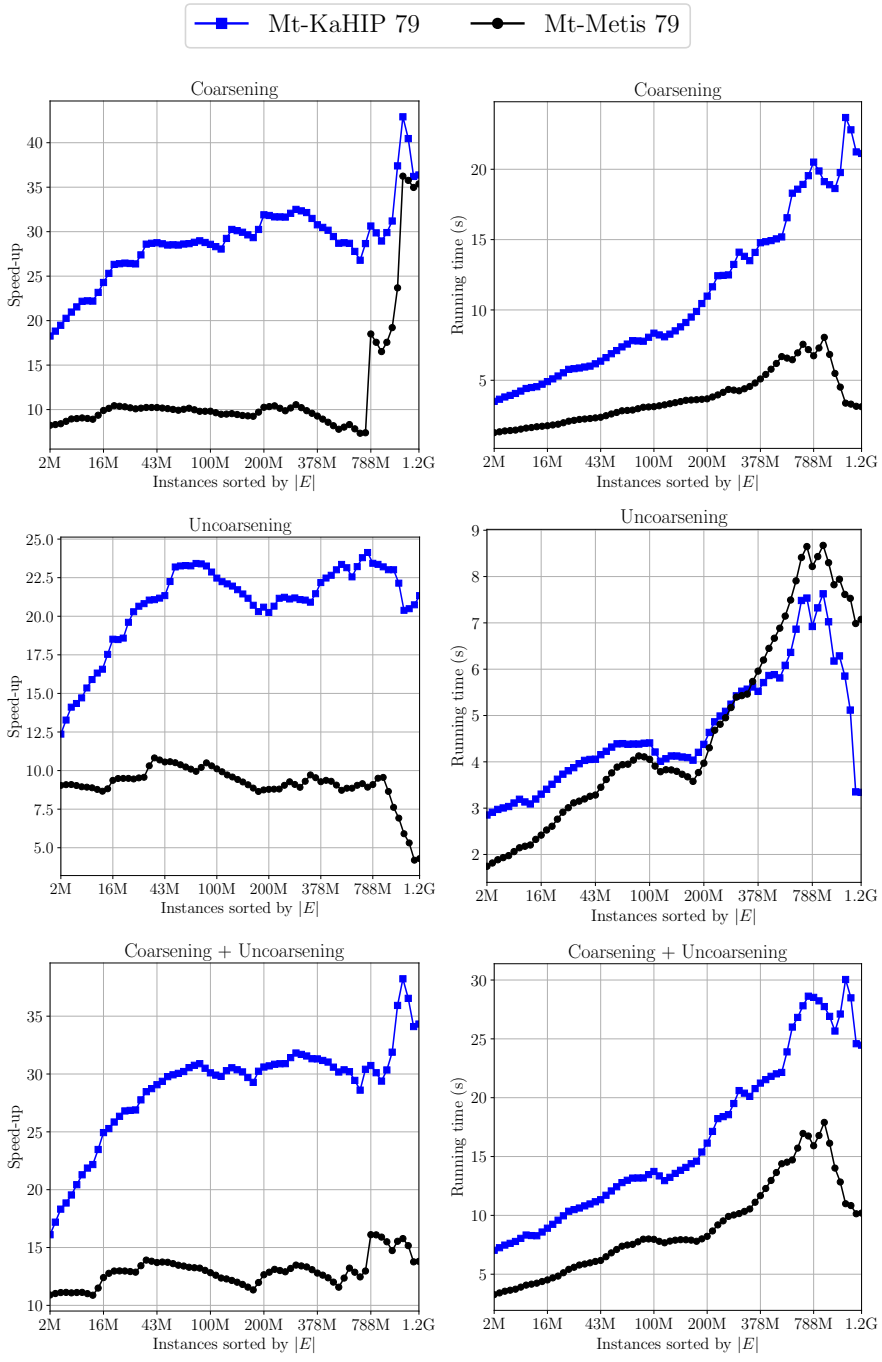


Figure 4.22: Scatter plots with speed-ups and average running times per edge of the different components for $p = 79$.


Mt-KaHIP 79
Mt-Metis 79
Coarsening
Speed-up
Running time (s)
Uncoarsening
Coarsening + Uncoarsening
2M
16M
43M
100M
200M
378M
788M
1.2G
Instances sorted by |E|

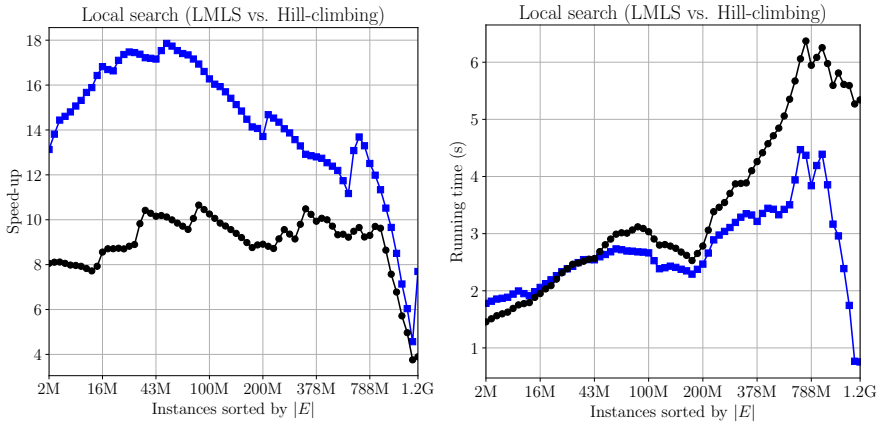
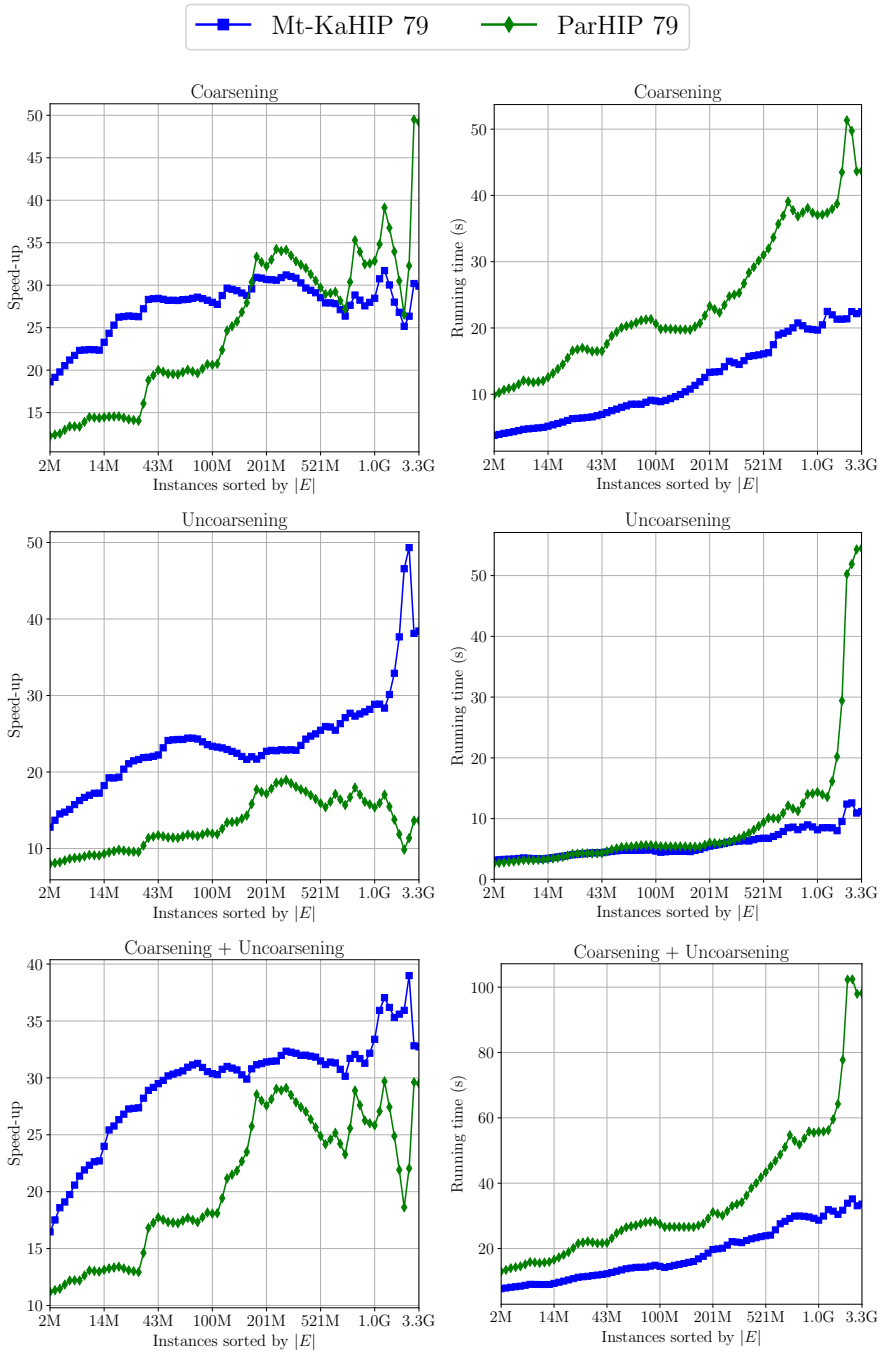


Figure 4.23: *Cumulative* harmonic mean speed-ups and geometric mean running times of the components of `Mt-KaHIP` and `Mt-Metis` for $p = 79$ on the set of instances $\mathcal{S}_{\texttt{Mt-Metis}}$. Here a point (x, y) means that the harmonic mean speed-up (geometric mean running time) of the graphs with $|E| \geq x$ is y.


Mt-KaHIP 79
ParHIP 79
Coarsening
Speed-up
Instances sorted by |E|
Coarsening
Running time (s)
Instances sorted by |E|
Uncoarsening
Speed-up
Instances sorted by |E|
Uncoarsening
Running time (s)
Instances sorted by |E|
Coarsening + Uncoarsening
Speed-up
Instances sorted by |E|
Coarsening + Uncoarsening
Running time (s)
Instances sorted by |E|
2M 14M 43M 100M 201M 521M 1.0G 3.3G

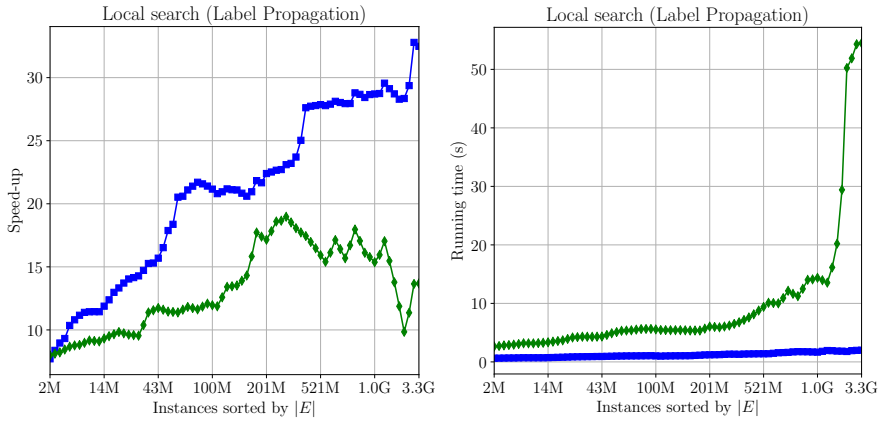


Figure 4.24: Cumulative harmonic mean speed-ups and geometric mean running times of the components of `Mt-KaHIP` and `ParHIP` for $p = 79$ on the set of instances $\mathcal{I}$. Here a point (x, y) means that the harmonic mean speed-up (geometric mean running time) of the graphs with $|E| \geq x$ is y.

Additional Performance Comparisons

Here we present additional performance comparisons of `Mt-KaHIP` to `Mt-KaHIP eco` and `PuLP`.

Performance comparison to `Mt-KaHIP eco`. We additionally compare `Mt-KaHIP` to `Mt-KaHIP eco`, which uses the local max matching algorithm during coarsening, on all 16 mesh type graphs (32 instances) from the set of instances $\mathcal{I}$. We consider `Mt-KaHIP eco` on the mesh type graphs since they do not have communities which the label propagation algorithm searches for during the coarsening phase of `Mt-KaHIP`. Note that matching algorithms are not biased towards communities. Therefore, we investigate how the parallel local max matching algorithm during the coarsening phase affects the resulting performance of our framework.

Figure 4.25 shows the scatter plots with speed-ups and running times per edge of the frameworks. `Mt-KaHIP` has better speed-ups than `Mt-KaHIP eco` and comparable times per edges with increasing number of edges. The harmonic mean speed-ups of `Mt-KaHIP` and `Mt-KaHIP eco` are 10.3 and 9.2, respectively. The geometric mean running times of `Mt-KaHIP` and `Mt-KaHIP eco` are 20.2 s and 21.5 s, respectively. `Mt-KaHIP eco` has worse speed-ups since some components of its coarsening phase are sequential but we are planning to parallelize them in future research.

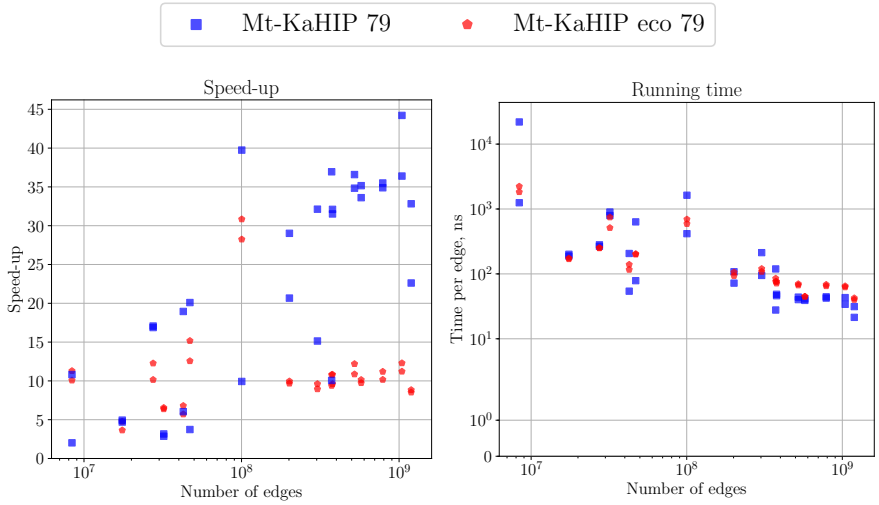


Figure 4.25: Scatter plots with speed-ups and average running times per edge of `Mt-KaHIP` and `Mt-KaHIP eco` for $p = 79$.

Performance Comparison to PuLP. We present performance comparisons of complete runs of Mt-KaHIP, Mt-KaHIP fast, and PuLP on machine A. Figure 4.26 shows the scatter plots with speed-ups and running times per edge of the frameworks. Mt-KaHIP fast has better speed-ups than PuLP. Note that with increasing number of edges, Mt-KaHIP has also better speed-ups than PuLP. To analyze the overall performance of the frameworks, we consider harmonic mean speed-ups and times per edge presented on Figure 4.27.

Table 4.14 shows the harmonic mean speed-ups and the geometric mean running times of the frameworks. Mt-KaHIP has worse harmonic mean speed-up and geometric running time since the coarsest graph is not small enough and the initial partitioning phase becomes a bottleneck. However, note that Mt-KaHIP fast does not have this problem and has better harmonic mean speed-up than PuLP. Although Mt-KaHIP has worse speed-ups than PuLP, Figure 4.27 shows that if we exclude the graphs with less than 47M edges then the harmonic mean speed-up of Mt-KaHIP is better than that of PuLP. In summary, although on average our algorithm is slower than PuLP,our framework has better quality (see Section 4.5.2) and we consider this as a fair trade off between quality and the running time.

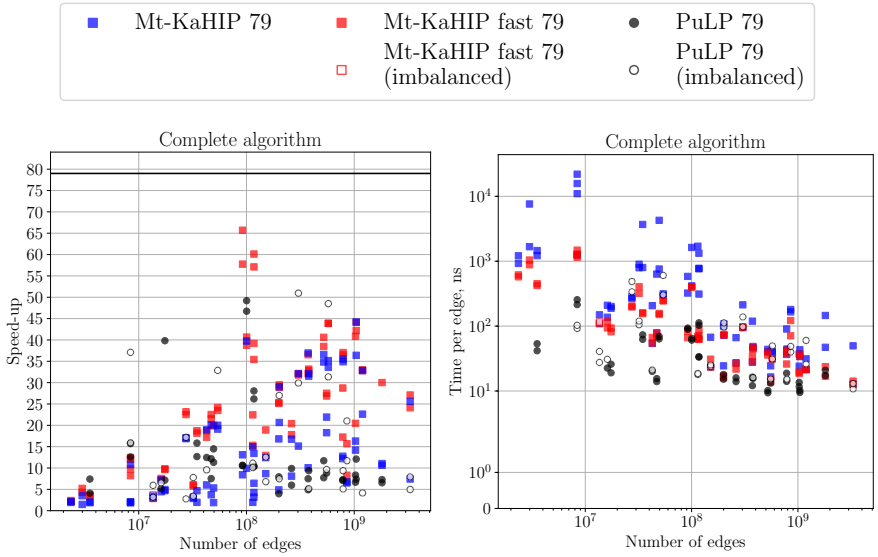


Figure 4.26: Scatter plots with speed-ups and average running times per edge of the frameworks for $p = 79$.

Table 4.14: Harmonic mean speed-ups and geometric mean running times of the frameworks over different sets of instances. Here $\mathcal{S}_{\texttt{PuLP}}$ *balanced* is 40 instances from $\mathcal{S}_{\texttt{PuLP}}$ that are partitioned without imbalance.

Algorithm	Speed-up		Running time (s)	
	$\mathcal{S}_{\texttt{PuLP}}$	$\mathcal{S}_{\texttt{PuLP}}$ *balanced*	$\mathcal{S}_{\texttt{PuLP}}$	$\mathcal{S}_{\texttt{PuLP}}$ *balanced*
`Mt-KaHIP`	6.6	5.8	23.7 s	27.6 s
`Mt-KaHIP fast`	**15.6**	**17.5**	10.1 s	8.7 s
`PuLP`	8.8	9.0	**5.1** s	**3.6** s

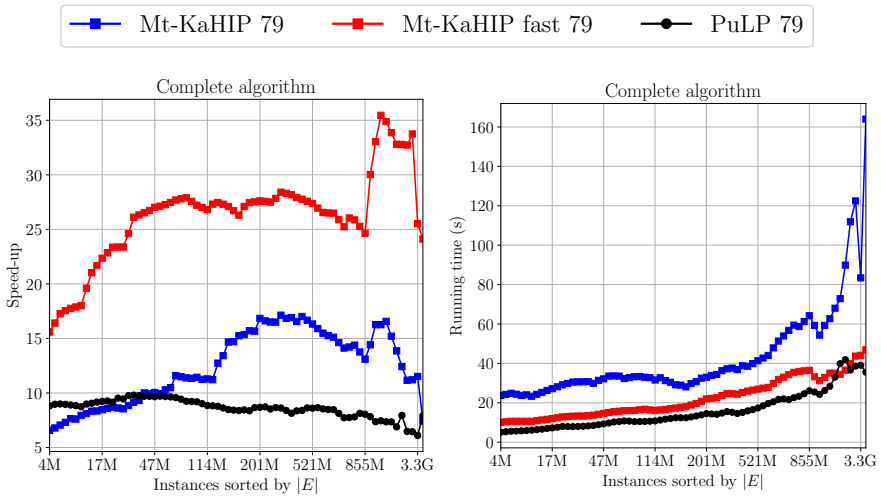


Figure 4.27: *Cumulative* harmonic mean speed-ups and geometric running times of the frameworks for $p = 79$ on the set of instances $\mathcal{S}_{\texttt{PuLP}}$. Here a point (x, y) means that the harmonic mean speed-up (geometric running time) of the graphs with $|E| \geq x$ is y.

4.5.4 Memory consumption

We now look at the memory consumption of `Mt-KaHIP`, `Mt-Metis`, and `ParHIP` on eight biggest graphs of the benchmark set $\mathcal{I}$ for $k = 16$ (for $k = 64$ they are comparable) on machine A. Figure 4.28 and Table 4.15 present memory consumptions for the frameworks with $p = 1$ and $p = 79$. We observe only small memory overheads of `Mt-KaHIP` when increasing the number of PEs. We explain these by the fact that all data structures created by each PE are either of small memory size (copy of a coarsened graph) or the data is distributed between them approximately uniformly (a hash table in LMLS). Note that all frameworks have relatively little memory overhead for parallelization. However, on average `Mt-KaHIP` consumes 33.1% less memory than `Mt-Metis` and 56.2% less memory than `ParHIP` on these graphs using 79 PEs.

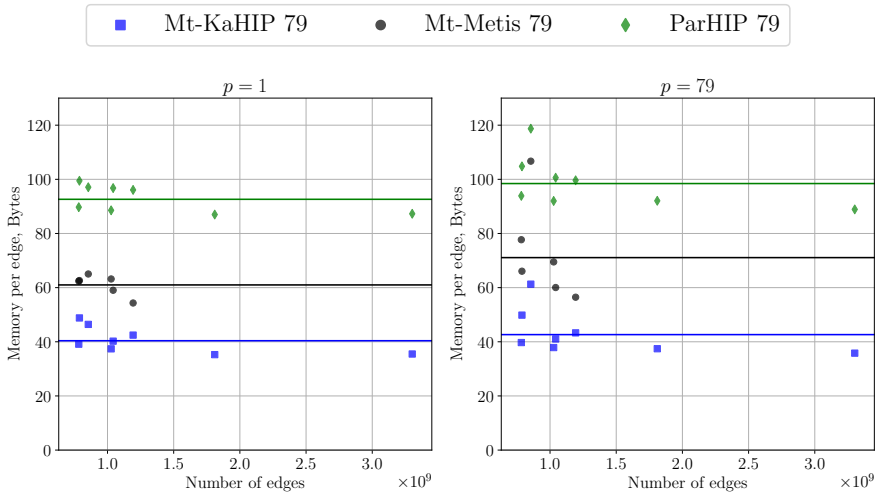


Figure 4.28: Memory consumption per edge in bytes. The horizontal lines are the geometric mean memory consumptions per edge.

Table 4.15: Memory consumption in gigabytes.

	$p = 1$			$p = 79$		
Graph	`Mt-KaHIP`	`Mt-Metis`	`ParHIP`	`Mt-KaHIP`	`Mt-Metis`	`ParHIP`
uk-2005	28.5	45.5	65.4	29.0	56.7	68.5
rgg_2_27_3d	35.8	45.9	73.0	36.6	48.5	76.9
webbase-2001	37.0	51.8	77.3	48.8	85.0	94.5
it-2004	35.8	60.5	84.7	36.2	66.5	88.0
del_2_27_3d	39.1	57.3	94.0	39.8	58.3	97.7

rgg_2_27	47.2	60.4	106.8	48.1	62.7	110.7
sk-2005	59.4	–	146.6	63.1	–	155.2
uk-2007	109.1	–	268.4	110.1	–	273.4
Geometrical mean	44.8	53.2	102.7	47.3	62.0	109.1

4.5.5 Influence of MGP Phases of Mt-KaHIP

We now analyze how the parallelization of the different phases affects solution quality of partitioning and present the speed-ups of each phase. We perform experiments on machine A with configurations of Mt-KaHIP in which only one of the phases (coarsening, initial partitioning, uncoarsening) is parallelized. The respective parallelized phase of Mt-KaHIP uses 79 PE and the other phases run sequentially. Running Mt-KaHIP with parallel coarsening increases the geometric mean of the cut by 0.68%, with parallel initial partitioning decreases the geometric mean of the cut by 1.86%, and with parallel local search increases the geometric mean of the cut by 0.57%. Furthermore, the Wilcoxon signed-rank test (see Section 2.3.3) shows that the difference between the partitions computed by Mt-KaHIP with a single PE and partitions computed by Mt-KaHIP with any parallel phase is statistically insignificant. Specifically, the p-values of the signed-rank tests for Mt-KaHIP with a single PE and Mt-KaHIP with only parallel coarsening, initial partitioning, or uncoarsening are 0.92, 0.19, and 0.52, respectively. Therefore, compared to Mt-KaHIP with a single PE, Mt-KaHIP with any parallel phase either does not affect solution quality significantly or improves the cut slightly on average. The parallelization of initial partitioning gives better cuts since it computes more initial partitions than the sequential version.

To show that the parallelization of each phase is important, we consider running time ratios and speed-ups of the phases of Mt-KaHIP and Mt-KaHIP fast when $p = 79$ (when all phases are parallel). Figure 4.29 and 4.30 show the corresponding running time ratios of the phases for the set of instances $\mathcal{I}$. The experiments were performed on machine A. The coarsening, initial partitioning, and uncoarsening phases of Mt-KaHIP have 7.0%, 10.3%, and 10.2% harmonic mean running time ratios. The coarsening, initial partitioning, and uncoarsening phases of Mt-KaHIP fast have 40.0%, 1.9%, and 31.7% harmonic mean running time ratios. We observe that each phase has the largest running time ratio for at least one instance. For the graph rgg_2_27 and $k = 16$, the *coarsening phase* of Mt-KaHIP takes 83.6% of the running time and its parallelization gives a speed-up of 36.0 and a full speed-up of 32.8. For the graph rgg_2_27_3d and $k = 16$, the *coarsening phase* of Mt-KaHIP fast takes 85.7% of the running time and its parallelization gives a speed-up of 39.6 and a full speed-up of 37.1. For the graph hollywood-2011 and $k = 64$, the *initial partitioning phase* of Mt-KaHIP takes 98.2% of the running time and its parallelization gives a speed-up of 1.3 and the overall speed-up is 2. For the graph webbase-2001 and $k = 16$, the *initial partitioning phase* of Mt-KaHIP fast takes 65.8% of the running time and its parallelization gives a speed-up of 0.18 and the overall speed-up is 8.3. Initial partitioning has a speed-up that is less than one on this graph since the coarsest graph computed with $p = 79$ has more edges than the coarsest graph computed with $p = 1$. For the graph er_2_22_2_23 and $k = 16$, the *uncoarsening phase* of Mt-KaHIP takes 71% of the running time and its parallelization gives a speed-up of 8.5 and the overall speed-up is 10.8. For the graph ba_2_22 and $k = 16$, the *uncoarsening phase* of Mt-KaHIP fast takes 84.3% of the running time and its parallelization gives a speed-up of 6.4 and the overall speed-up

is 8.2. Uncoarsening has a low speed-up on these graphs since the running times of `FindMoves` and `ApplyMoves` in LMLS are comparable and `ApplyMoves` is sequential. The harmonic mean speed-ups of the coarsening phase, the initial partitioning phase and the uncoarsening phase of `Mt-KaHIP` are 18.6, 1.0 and 12.8, respectively. The harmonic mean speed-ups of the coarsening phase, the initial partitioning phase and the uncoarsening phase of `Mt-KaHIP fast` are 16.4, 1.0 and 10.9, respectively.

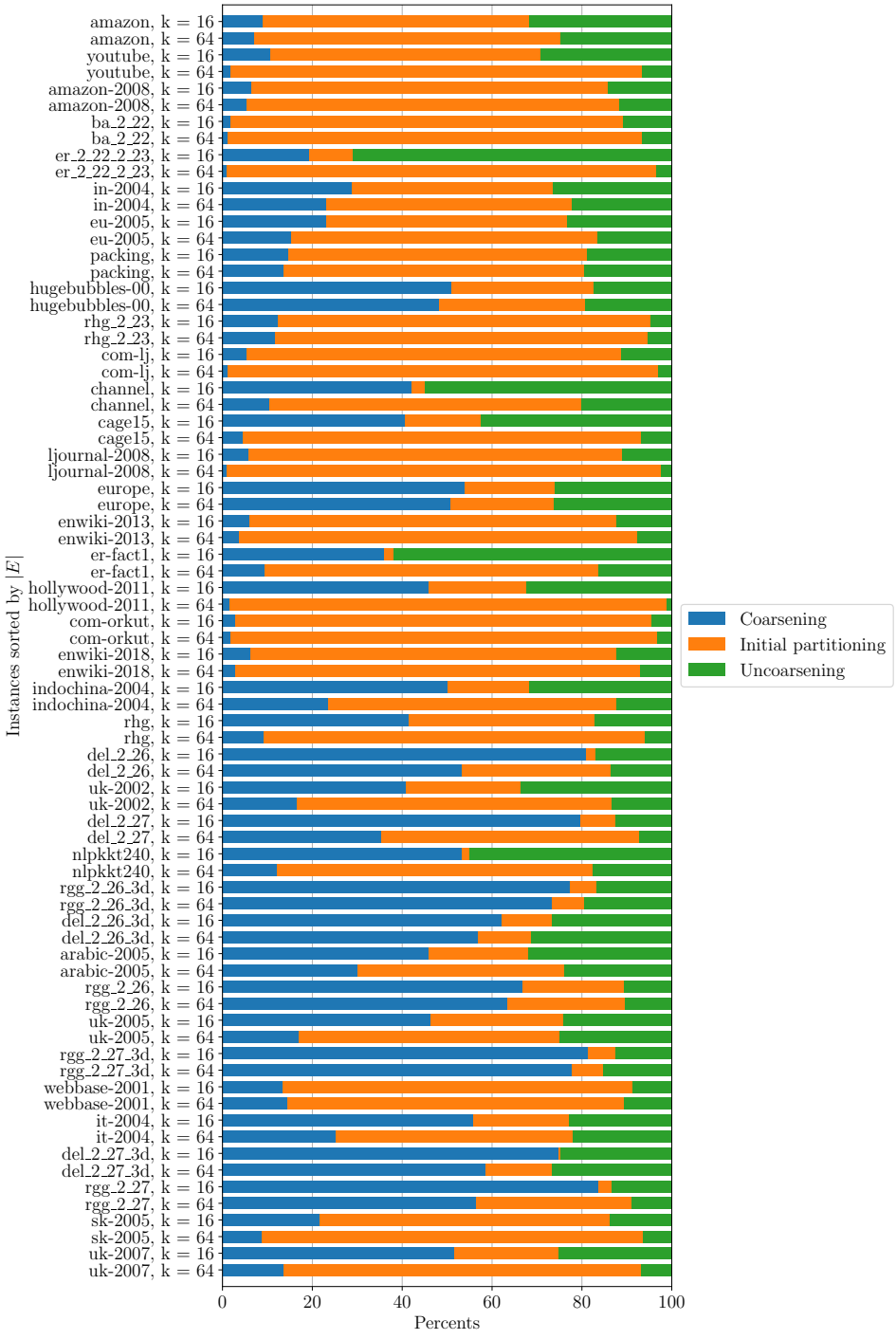


Figure 4.29: Running time ratios of components of `Mt-KaHIP`.

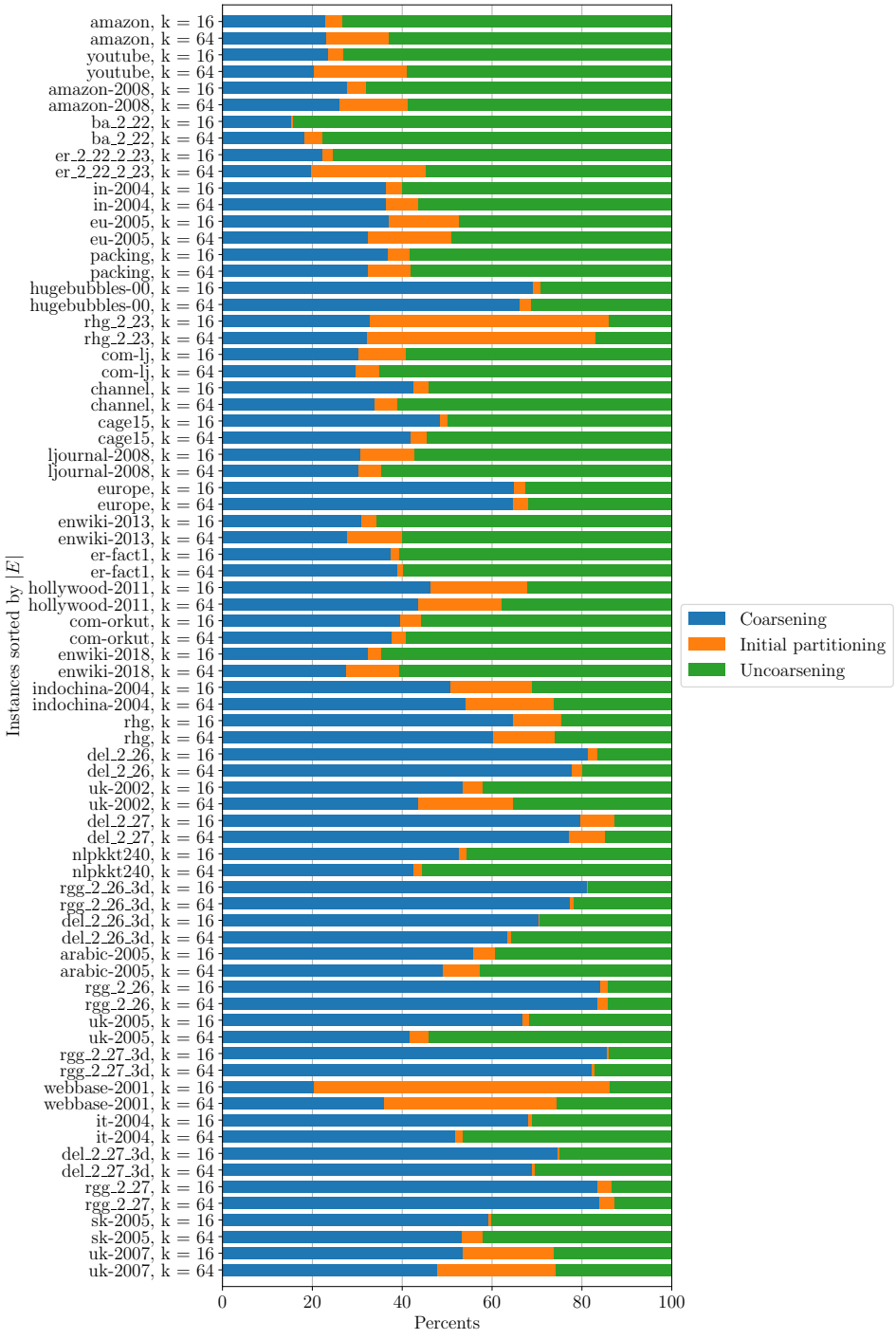


Figure 4.30: Running time ratios of components of `Mt-KaHIP fast`.

4.6 Conclusion and Future Work

Graph partitioning is a key prerequisite for efficient large-scale parallel graph algorithms. We presented an approach to multi-level shared-memory parallel graph partitioning that guarantees balanced solutions, shows high speed-ups for a variety of large graphs and yields very good quality independently of the number of cores used. Previous approaches have problems with recently grown structural complexity of networks that need partitioning – they often show a negative trade-off between speed and quality. Important ingredients of our algorithm include parallel label propagation for both coarsening and refinement, parallel initial partitioning, a simple yet effective approach to parallel localized local search. Considering the good results of our algorithm, we want to further improve it. More specifically, we are planning to further improve scalability of parallel coarsening, parallel initial partitioning and parallel LMLS. An interesting problem is how to apply moves in Section 4.3.3 without the gain recalculation. The solution of this problem will increase the performance of parallel LMLS. One possible solution is to construct a direct acyclic graph (DAG) which represents dependencies between moves. Each vertex in the DAG corresponds to a move of a vertex in the original graph. Further, if a PE $i < j$ moves a vertex that is adjacent to a vertex moved by the PE j then these vertices are connected in the DAG. Additionally, there are also directed edges between vertices that are moved by a single PE (an edge is directed from the first moved vertex to the second moved vertex) if they are connected in the original graph. Each PE can process a corresponding subgraph of the DAG in topological order taking into account dependencies between subgraphs which are represented by edges that run between subgraphs. Further quality improvements should be possible by integrating a parallel version of the flow based techniques used in `KaHIP`. It would be interesting to try reinforcement learning techniques [SB11] to improve LMLS. There are two main aspects of LMLS which can be improved with reinforcement learning are: deciding which vertices to move; estimating a possible gain increase and, thus, deciding whether to stop LMLS or not.

Moreover, we want to investigate the influence of other parallel clustering algorithms – different from label propagation – on quality and scalability of graph partitioning. Specifically, there are numerous different parallel clustering algorithms that optimize different measures (e.g., modularity) and can potentially improve scalability and quality of graph partitioning. For example, Meyerhenke and Staudt [SM16] suggested a parallel clustering algorithm that optimizes modularity; Shun et al. [Shu+16] suggested a parallel clustering algorithm that optimizes conductance of clusters.

Another interesting problem is the theoretical and extensive practical research of cache-aware hash tables. From the theoretical point of view, it is important to show how additional collisions affect the resulting time complexity. Our hope is that it remains $\mathcal{O}(1)$. The practical research should include experiments of cache-aware hash tables with keys generated using different distributions. Furthermore, recently a machine learning approach for search was suggested by Kraska et al. [Kra+18].

The idea is to learn a machine learning formula that encapsulates the knowledge about the distribution of keys. This machine learning formula can be used as a hash function. Note that it is possible if the distribution of keys is static. It is interesting to investigate if this machine learning technique can be used in a locality preserving hash table.

Chapter 5

5 (Semi-) External Multi-level Graph Partitioning

Graph partitioning and clustering problems are often solved to analyze or process large graphs in various contexts such as social networks, web graphs, road networks, or in scientific numeric simulations. To be able to process huge unstructured networks on cheap commodity machines, one can partition the graph under consideration into a number of blocks such that each block fits into the internal memory of the machine while edges running between blocks are minimized (see for example [KBG12]). However, to do so the partitioning algorithm itself has to be able to partition networks that do not fit into the internal memory of a machine.

Reference. This chapter is based on the conference paper [ASS15] published together with Peter Sanders and Christian Schulz. The text was mainly written by Yaroslav Akhremtsev and Christian Schulz with editing by Peter Sanders. The design and analyses of the algorithms were made by all authors. The algorithms were implemented by Yaroslav Akhremtsev.

Contribution: In this chapter, we present semi-external and external algorithms for the graph partitioning problem that compute high-quality solutions. Our MGP algorithms perform coarsening and uncoarsening using external memory. Our approach uses a (semi-)external label propagation algorithm that rapidly shrinks large complex networks during coarsening. Furthermore, we describe a (semi-)external contraction algorithm. We again use (semi-)external label propagation during uncoarsening to refine graphs. Note that only the semi-external label propagation algorithm provides guarantees on maximum sizes of clusters. Thus, we develop another external memory clustering algorithm that constructs clusters of sizes that do not exceed a preset threshold.

The chapter is organized as follows. After discussing related work in Section 5.1, we explain how clusterings can be used to build a graph hierarchy to be used in a (semi-)external multi-level algorithm in Section 5.2. Moreover, it presents the *first* external memory algorithm to tackle the graph partitioning problem. Next, Sections 5.2.1, describes a (semi-)external label propagation algorithm that computes a clustering with or without a size constraint and the parallelization of the semi-external version. Further, Section 5.2.2 describes the (semi)-external graph clustering

algorithm based on graph coloring. In contrast to the external label propagation algorithm, the external color-based algorithm constructs a *size-constrained* clustering. Experiments to evaluate performance of our algorithms are presented in Section 5.3. Finally, we conclude in Section 5.4.

5.1 Related Work

The detailed description of external memory model is described in Section 2.2.2. In this chapter, we use the time forward processing technique [Chi+95; Zeh02]. The main idea is that if computations can be represented using a directed acyclic graph then they can be efficiently performed using an external memory priority queue. Specifically, if one computation occurs at time t_1 and another computation occurs at time $t_2 (t_1 < t_2)$ and uses the result of the first computation then we can send the result using an external priority queue. Specifically, we insert the result of the first computation with the key t_2 into the priority queue and extract it at time t_2.

The first external priority queue was presented by Agre [Arg03a] that allows to extract the minimum element and to insert a new element using amortized $\mathcal{O}(1/B \log_{M/B} N/B)$ I/Os. In our thesis, we use an external priority queue by Sanders [San99] since there is a fast implementation of it in the STXXL library [Bec+].

Graph clustering with the label propagation algorithm (LPA) has originally been described by Raghavan et al. [RAK07]. See Section 3.2.2 for more details. Meyerhenke et al. [MSS14] introduced the size-constrained LPA. Furthermore, they use this algorithm during the coarsening and uncoarsening phases of the MGP scheme to compute graph partitions of large complex networks. In this chapter, we present a semi-external and an external versions of this algorithm. There are other semi-external algorithms to tackle the graph partitioning problem which are based on streaming [SK12]. However, they do not use the MGP approach and do not achieve high-quality solutions. To the best of our knowledge, our algorithm is the first that tackles the graph partitioning problem in the external memory model. We compare our algorithm to two state of the art graph partitioning frameworks: `KaHIP` and `kMetis` that are described in Section 3.8.

5.2 (Semi-)External MGP

We now present the main ingredients we use to obtain a (semi-)external multi-level graph partitioning (MGP) algorithm. The details of the MGP scheme are described in Section 3.1. First of all, we outline a (semi-)external algorithm to create graph hierarchies for the MGP scheme. Meyerhenke et al. [MSS14] create a graph hierarchy in the internal memory by iteratively contracting size-constrained graph clusterings

that are obtained using the label propagation algorithm (LPA). The contraction of a clustering works as follows: each cluster is contracted into a single vertex. The weight of the vertex is set to the sum of the weights of all vertices in the original cluster. There is an edge between two vertices A and B in the contracted graph if the two corresponding clusters are adjacent to each other in G. The weight of an edge (A, B) is set to the sum of the weight of the edges that run between A and B. Due to the way contraction is defined, a partition of the coarse graph corresponds to a partition of the finer graph with the same cut and balance. We refer to the contracted graph as quotient graph.

Cluster contraction is an aggressive coarsening strategy. In contrast to most previous approaches, it can drastically shrink the size of irregular networks. The intuition behind this technique is that a clustering of the graph (one hopes) contains many edges running inside the clusters and only a few edges running between clusters, which is favorable for the edge cut objective. Regarding complexity, experiments in [MSS14] indicate that already one contraction step can shrink the graph size by orders of magnitude and that the average degree of the contracted graph is smaller than the average degree of the input network. Thus, it is very likely that the graph will fit into the internal memory after the first contraction step. On the other hand, by using a different size constraint ($|V_i| \leq L_{\max} := (1+\epsilon)\lceil c(V)/k \rceil$), the LPA can also be used as a simple strategy to improve solution quality during uncoarsening.

To obtain a (semi-)external MGP algorithm, we develop a (semi-)external variant of the size-constrained LPA, the contraction algorithm, and the projection algorithm that projects a partition of a coarse graph to the corresponding finer graph. By doing this, we have a (semi-)external algorithm that constructs a graph hierarchy and projects a partition to finer levels. Once the graph is small enough to fit into the internal memory, we use the KaHIP framework to compute a partition of the graph. Additionally, we use a (semi-)external size-constrained LPA as a local search algorithm to improve the solution on the finer levels that do not fit into the internal memory. We proceed by explaining how graph clustering can be obtained in both, the semi-external and the external memory models. In Section 5.2.1, we present a semi-external size-constrained LPA and an external LPA that does not use size constraints. Furthermore, we explain how both algorithms can be parallelized. In Section 5.2.2, we describe a coloring-based graph clustering algorithm inspired by label propagation that is able to maintain size constraints in the external memory model. Finally, we explain an external contraction in Section 5.2.3 and an external projection of partitions in Section 5.2.4.

5.2.1 Label Propagation Clustering

Label propagation works as follows. In the beginning, each vertex belongs to its own cluster. and the algorithm works in rounds. During each round, the algorithm visits each vertex in increasing order of their IDs. When a vertex v is visited, it is *moved* to the block with the strongest connection to v, that is, it is moved to the cluster V_i that

maximizes $\omega(\{(v,u) \mid u \in N(v) \cap V_i\})$. Ties are broken randomly. If the algorithm is used to compute a size-constrained clustering, the selection rule is modified such that only moves are eligible that do not result in overloaded blocks.

Suppose the algorithm is currently processing a vertex v. We scan the adjacency list of the vertex v and compute the new cluster of this vertex. If there is a size constraint, we also need a scheme to manage the sizes of each cluster/block.

Active Vertex Strategy. The active vertex strategy can be used to speed up computations of the LPA [SM16]. The basic idea is that after the first round of the algorithm, a vertex changes its cluster only if at least one of its neighbors changed its cluster in the previous round. The active vertex strategy keeps track of vertices that can potentially change their cluster. Specifically, a vertex is called *active* if at least one of its neighbors changed its cluster in the *previous round*.

To translate this into our computational model, we additionally keep the set of active vertices and calculate new cluster IDs only for these vertices. More precisely, in the beginning all vertices are active. After each round the algorithm updates the set of active vertices by inserting the neighbors of vertices which have changed their cluster and by deleting vertices whose neighbors have not changed their cluster. In the worst case the computational complexity of the LPA remains the same. However, our experiments indicate that the active vertex strategy decreases the computational time of the LPA given a sufficient number of rounds.

To implement the label propagation algorithm with active vertex strategy, we use two priority queues (external or internal, depending on the variant of the LPA). Both priority queues contain active vertices: the first priority queue contains active vertices that will be processed in the current round and the second priority queue contains vertices which will be processed in the next round. If a vertex v changes its cluster then we push all its neighbors with larger IDs to the first priority queue and the neighbors with smaller IDs to the second priority queue. We use priority queues because the algorithm processes vertices in increasing order of their IDs. In the external LPA, we store for each edge (v,u) in the array of edges additional information about the cluster of u. This allows us to detect if the cluster of a vertex changed. We add cluster$[u]$ in the array of edges in the beginning of algorithm using $\text{Sort}(|E|)$ I/O operations and maintain them up-to-date during the course of the algorithm.

Semi-External Label Propagation

This is the simple case: since we have $\mathcal{O}(|V|)$ internal memory, we can afford to store the cluster IDs in the internal memory. Additionally, we maintain an array of size $|V|$ in the internal memory that stores the cluster sizes. Hence, one iteration of the semi-external LP algorithm can be done using $\text{Scan}(|E|)$ I/O operations by iterating over the external array of edges.

Parallel Semi-External Label Propagation. Recall that the LPA iterates over the external array of edges. In order to accelerate the semi-external LPA and to get closer to the I/O bound, we parallelize the processing of a disk block of edges. We divide the disk block into equal ranges and process them in parallel (see Figure 5.1 for an example). Each PE t owns a range $[begin_t; end_t)$ of a disk block to process. But before processing the range, PE t shifts the range so that each adjacency list in the block is scanned by exactly one PE. Specifically, if $begin_t$ is not at the beginning of the current adjacency list then PE t skips it. Note that the PE that processes the beginning of the disk block does not shift since if an adjacency list runs between two disk blocks then the PE must continue to scan it during the processing of the next block. Consider the example depicted in Figure 5.1. Here, the PE finds the end of the adjacency list 1 in $[begin_t; end_t)$ and iterates through the elements until the end of adjacency list 2 is reached. The shaded area in Figure 5.1 represents the range that will be actually processed by PE t.

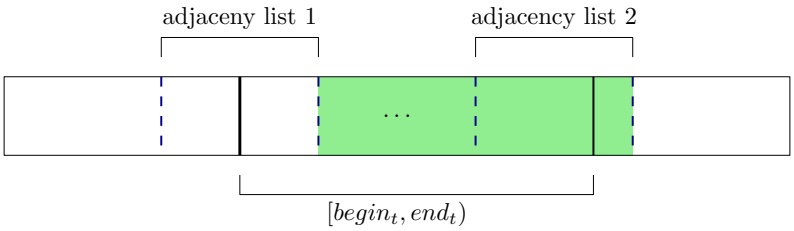
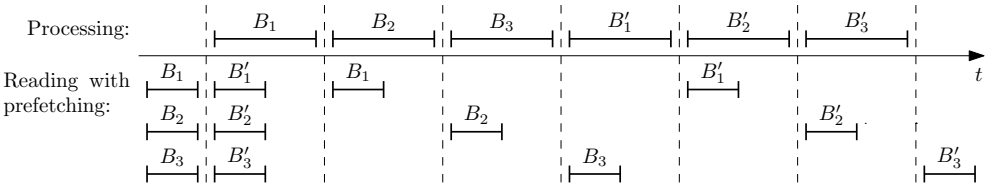


Figure 5.1: Range processed by a PE t. The PE t skips the adjacency list 1 and PE $t-1$ will scan it. This allows to avoid the situation when the same adjacency list is scanned by several PEs.

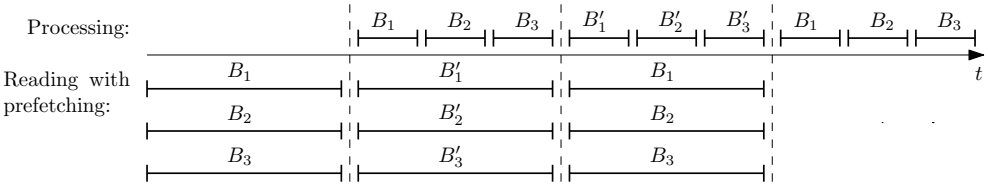
To maintain up-to-date cluster sizes, we do not move a vertex immediately, i.e. we store the moves which were generated by the PEs during the processing of the disk block. Afterwards, all moves are processed sequentially and we make a move if it does not violate the size constraint.

To show that decreasing processing time of a block is important, we consider two cases. The first case is when the time to process a block t_p is greater than the time to read a block from a disk t_r. The second case is the opposite; that is, $t_p < t_r$. Figures 5.2 (a), 5.2 (b) show time plots for both cases. Let D denotes the number of disks and each disk has two corresponding blocks in the internal memory to which it reads data using prefetching. We can see that in the first case all blocks are read from disks sequentially since each disk waits until one of its two corresponding blocks is processed and can be overwritten. Although the overall number of I/O operations is $|E|/B$ for both cases, their running times differ significantly. In the first case, the running time is $\mathcal{O}(|E|/B \cdot t_p) = \mathcal{O}(|E|/B \cdot ct_r)$ given $t_p = ct_r,\ c > 1$. In contrast, in the second case processing and reading takes only $\mathcal{O}(|E|/B \cdot t_p) = \mathcal{O}(|E|/(B \cdot c) \cdot t_r)$

time in the second case given $t_p = t_r/c$. Thus, if we are able to process a block D times faster than reading it then the running time is $\mathcal{O}(|E|/(B \cdot D) \cdot t_r)$.



(a) The first case when the time to process a block is greater than the time to read a block ($t_p > t_r$).



(b) The second case when the time to read a block is greater than the time to process a block ($t_p < t_r$).

Figure 5.2: Timeline of when disk blocks are prefetched and processed. Here $D = 3$. B_1, B'_1 are blocks of disk 1. B_2, B'_2 are blocks of disk 2. B_3, B'_3 are blocks of disk 3.

External Label Propagation

To perform the LPA in the external memory model, we use time forward processing [Chi+95; Zeh02] to propagate cluster IDs of adjacent vertices. More precisely, we maintain two external priority queues [San99]: one for the current and one for the next round. Initially, the current priority queue contains triples (v, c, w) for each edge $(u, v) \in E : v < u$ where v is the key value, w denotes the weight of the edge and c is the current cluster ID of u. When the algorithm scans a vertex v, all triples (v, c, w) are on the top of the current priority queue since the vertices are processed in increasing order of their IDs. The tuples are then extracted using the operations `Pop` and `Top`. This means, we know the current cluster ID of all adjacent vertices of u and can calculate the new cluster ID. After the new cluster ID is computed, the algorithm pushes triples with the new cluster ID for all adjacent vertices into the next and current priority queue depending on the vertex ID of the neighbor v: if $u < v$ we push $(v, \text{cluster}[u], w(u, v))$ to the current priority queue and if $v < u$ we push it to the next priority queue. At the end of a round we swap the priority queues.

Each operation of the priority queue `Pop`, `Push` and `Top` is called $\mathcal{O}(|E|)$ times and can

be done using $\mathcal{O}(1/B \log_{M/B} |E|/B)$ I/O operations amortized [Arg03b; San99]. Thus, the overall algorithm uses $\mathcal{O}(|E|/B \log_{M/B} |E|/B) = \text{Sort}(|E|)$ I/O operations.

5.2.2 Coloring-based Graph Clustering

We now present another approach to cluster a graph in the external memory model. The algorithm is able to maintain the sizes of all clusters in the external memory model. The *main idea* of the algorithm is to process *independent sets* of vertices. Since vertices of an independent set are not connected, a change of the cluster ID of a vertex does not affect other vertices in the set. However, we need to take into account the changes of cluster IDs of neighbors.

Assume that we have a vertex coloring $C = \{C_1, C_2, \ldots, C_\ell\}$ of the graph, where C_i is the set of vertices with the same color i. We denote the color of a vertex v as $C[v]$. Note that each set C_i forms an independent set. For each set C_i, we maintain an external array (bucket) of tuples $\mathcal{T}_i$ and allocate a buffer of size B in the internal memory for each bucket $\mathcal{T}_i$. Here we assume that the number of colors $|C| = \mathcal{O}(M/B)$.

The bucket clustering algorithm works in rounds. Roughly speaking, in each round it processes the buckets in increasing order of their color and updates the cluster IDs of all vertices. When we process a bucket we need the cluster IDs of all vertices that are adjacent to the vertices in the bucket. Therefore, we define the content of a bucket as follows: we store tuples where each tuple contains the cluster ID of the corresponding neighbor, its color, ID of the neighbor and the weight of the edge. More precisely, initially for a vertex u, we store the following tuples for all neighbors v with $\mathrm{C}[v] < \mathrm{C}[u]$ in the corresponding buckets $\mathcal{T}_{\mathrm{C}[v]}$: $(v, \text{cluster}[u], u, w(u, v), \mathrm{C}[u])$. To do this efficiently, we augment the array of edges by adding the color of the target vertex v to each edge (u, v) before the initialization step. Note that these tuples contain the complete information about the graph structure and that the information suffices to update clusters of vertices.

When the algorithm processes a bucket $\mathcal{T}_i$, it sorts the elements of the bucket lexicographically by the first and second component. Afterwards, it scans the tuples of the bucket and calculates a new cluster ID for each vertex in C_i in the same manner as the LPA. After the bucket is processed, we push tuples with the new cluster IDs to the corresponding buckets; that is, for each tuple $(v, \text{cluster}[u], u, w(v, u), \mathrm{C}[u])$ in bucket $\mathcal{T}_i$, we push the tuple $(u, \text{cluster}[v], v, w(u, v), \mathrm{C}[v])$ into the bucket $\mathcal{T}_{\mathrm{C}[u]}$. Finally, we clear the bucket.

Lemma 5.1

Processing a bucket $\mathcal{T}$ requires Sort($|\mathcal{T}|$) I/O-operations.

Proof. For sorting the bucket, we need $\text{Sort}(|\mathcal{T}|)$ I/O operations. We need $\text{Scan}(|\mathcal{T}|)$ I/O operations for scanning the bucket. Hence, the algorithm uses $\text{Sort}(|\mathcal{T}|)$ I/O operations. □

Theorem 5.2

The bucket algorithm requires Sort($|E|$) I/O-operations for one iteration of label propagation.

Proof. Adding the information about the colors of neighbors to the edges requires Sort($|E|$) I/O operations. Our bucket initialization uses Scan($|E|$) I/O operations. Suppose we have the buckets $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$. All buckets can be processed using Sort($|\mathcal{T}_1|$)+ $\ldots$ + Sort($|\mathcal{T}_\ell|$) = Sort($|E|$) I/O operations. Overall, we use Scan($|E|$) I/O operations for pushing tuples with the new cluster IDs into the respective buckets. Hence, the algorithm can be implemented using Sort($|E|$) I/O operations.

Graph Coloring

Computing a graph coloring is an important part of the bucket graph clustering algorithm. Note that the number of colors is equal to the number of buckets and we want to maintain as few buckets as possible. This is due to the fact that we need $\mathcal{O}(B)$ internal memory space per bucket. Moreover, the size of each bucket must be smaller than an upper bound, since each bucket has to fit into the internal memory during our experiments. To compute a coloring, we use the time forward processing technique [Zeh02] with an additional size constraint on the color classes that can be maintained in the internal memory. This allows us to build a coloring using Sort($|E|$) I/O operations. Note that the coloring is computed only once so that the cost for computing the coloring can be amortized over many iterations of label propagation.

External Graph Clustering Algorithm with Size Constraints

In this section we describe how we modify the coloring-based clustering algorithm, so that it can handle a size constraint. The main advantage of the coloring-based clustering algorithm is as follows. When we process a bucket, the cluster IDs of all adjacent vertices will not change. This allows us to maintain a data structure with up-to-date sizes of the clusters of the vertices of the independent set and their neighbors. In the following, we consider two different data structures depending on whether a bucket fits into the internal memory or not. In both cases, we use an external array that stores the sizes of all clusters. We start by explaining the case where each bucket fits into the internal memory.

Case A: each bucket fits into the internal memory. In this case, we can use a hash table $\mathcal{H}$ to maintain the cluster sizes of the current bucket. The key of $\mathcal{H}$ is the cluster ID and the value is the current size of the cluster. When we process a bucket $\mathcal{T}_i$, the hash table $\mathcal{H}$ can be built as follows. We collect all cluster IDs of the vertices of the current independent set as well as their neighbors, sort them and then iterate through the external array to get the current cluster sizes. After finishing to

calculate a new cluster ID for each vertex in C_i, we write the updated cluster sizes to the external array. Hence, the cluster sizes are up-to-date after we processed the current bucket.

Theorem 5.3

The coloring-based clustering algorithm with size constraints uses $t \cdot Scan(|V|) + Sort(|E|)$ I/O operations, where $t = \max(|E|/M, |C|)$ is the amount of buckets such that each bucket fits into the internal memory.

Proof. First, we prove the complexity to create the data structure containing the clusters sizes. In the worst case, each tuple $(v, \text{cluster}[u], u, w(v,u), \text{C}[u])$ of the bucket $\mathcal{T}$ has a unique cluster ID and also the cluster ID of each vertex v is unique. Hence, we need an additional $\mathcal{O}(|\mathcal{T}|)$ internal memory space for the hash table. To save and sort the cluster IDs of vertices in the bucket, we use $\mathcal{O}(1)$ I/O operations since the bucket fits into the internal memory. Reading and writing the sizes of clusters to and from the external array require $\text{Scan}(|V|)$ I/O-operations.

Now we estimate the number of buckets that fit into the internal memory. There are two cases. If a bucket does not fit into the internal memory, we need to divide it into multiple buckets. Since the overall size of all buckets is $\mathcal{O}(|E|)$, the minimum number of buckets (such that each fits into the internal memory) is $\mathcal{O}(|E|/M)$. Otherwise, if all buckets fit into the internal memory, we have $|C|$ buckets. Since we want each bucket to fit into the internal memory, we have $\max(|E|/M, |C|)$ buckets.

The overall I/O-volume is estimated as follows: for each bucket we need $\text{Scan}(|V|)$ additional I/O-operations to create the data structure containing the sizes of the clusters. There are at most $\max(|E|/M, |C|)$ buckets. Hence, the total number of I/O-operations is $\max(|E|/M, |C|) \cdot \text{Scan}(|V|) + \text{Sort}(|E|)$ (to perform the main part of the bucket clustering algorithm). □

Case B: there is at least one bucket that does not fit into the internal memory. This case is somewhat more complicated, since we cannot afford to store the hash table in the internal memory. Basically, when we process a bucket $\mathcal{T}_i$, we do not use a hash table but an external priority queue and additional data structures which contain enough information to manage the cluster sizes. More precisely, we define a structure $\mathcal{M}$ that stores which vertices need the updated cluster size information if a vertex from the bucket changes its cluster ID. Vertices from C_i are still processed in increasing order of their IDs. We now explain these structures in detail.

For a vertex v in the current independent set C_i, let $C(v) := \{\text{cluster}[u] \mid (v,u) \in E\} \cup \{\text{cluster}[v]\}$ denote the set of adjacent clusters. An example is shown in Figure 5.3. These are the clusters that can possibly change their size if v changes its cluster. We now need to find all vertices from the independent set that are adjacent to these clusters or belong to these clusters because they need to receive the updated cluster size.

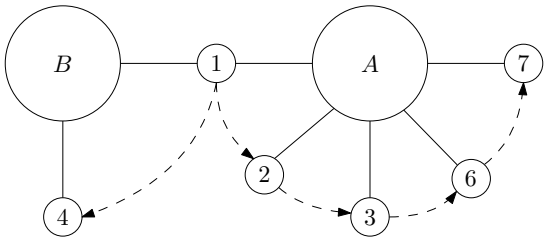


Figure 5.3: In this example, $C(1) = \{A, B, \text{cluster}[1]\}$ and $C(4) = \{B, \text{cluster}[4]\}$. Dashed lines denote forwarding cluster size changes. A, B are cluster IDs, 1-7 denote vertex ID. Vertex 6 belongs to cluster A. The sets of adjacent vertices for the cluster A and B are $N_A = \{1, 2, 3, 6, 7\}$ and $N_B = \{1, 4\}$. Moreover, $M_1 = \{(2, A), (4, B)\}$, $M_2 = \{(3, A)\}$.

The first additional data structure contains only vertices of the independent set. It is needed to build the next data structure $\mathcal{M}$. Let $N_c := \{u \in C_i \mid \exists (u, v) \in E : \text{cluster}[v] = c\} \cup \{u \in C_i \mid \text{cluster}[u] = c\}$ be the set of adjacent vertices for a cluster c that are in the current independent set (including the vertices that are in the cluster). We sort N_c in increasing order of vertex IDs and remove repeated elements. For a cluster c, the set N_c contains all vertices from the independent set that are adjacent to the cluster. Moreover, the order in N_c is the same to the processing order of the independent set. We denote the j-th vertex of N_c as N_c^j. The second additional data structure uses the first one and is defined as the set $M_v := \{(u, c) \mid c \in C(v),\ \exists j\ N_c^j = v,\ N_c^{j+1} = u\}$. It contains the vertices to which the vertex v must forward information about changes in the cluster sizes. Roughly speaking, for each cluster in the neighborhood of v (including the cluster of v), M_v contains the adjacent vertex of the cluster that will be processed next. This way the information can be propagated easily. An example is shown in Figure 5.3.

We now explain the details of the algorithm when processing one bucket. First, we compute the sets N_c. To do so, we build a list N of pairs (c, v) that are sorted lexicographically by their first and second component, where c is the ID of the cluster adjacent to v. For building this list, we iterate through the bucket and add pairs (c, v) for each tuple $(v, c, \ldots) \in \mathcal{T}_i$ to the list and also add the pair $(\text{cluster}[v], v)\ \forall v \in C_i$. Then we sort these pairs and we are done. Note that $|N| = |C_i| + |\mathcal{T}_i| = \mathcal{O}(|\mathcal{T}_i|)$. To compute the sets M_v, we build a list $\mathcal{M}$ of triples (v, c, u), where v is the vertex ID and $(u, c) \in M_v$. For each $N_c^j = v$ and $N_c^{j+1} = u$ the triple (v, c, u) is added to the list. Afterwards, the triples are sorted by the first component. Note that the size of the list is at most $\mathcal{O}(|\mathcal{T}_i|)$.

Recall, that the set M_v contains the vertices that have to receive the changes in the cluster sizes. To forward the information, we use an external priority queue. The priority queue contains triples (v, c, sz), where v is the vertex ID which also serves as key value, $c \in C(v)$ is the cluster and sz the size of the cluster. We initialize the

priority queue as follows: we iterate through the sets N_c and put the tuples (v_1, c, sz) in the priority queue, where v_1 is the first vertex in N_c. The sizes of the clusters are obtained from the external array containing the cluster sizes. Then the vertices are processed. After a vertex v is processed, we add (u, c, sz) to the priority queue for each pair $(u, c) \in M_v$. After we processed a bucket, we update the cluster sizes in the external array.

Lemma 5.4

When vertex v is processed there is a triple (v, c, sz) for each adjacent cluster on the top of priority queue with up-to-date cluster sizes.

Proof. Consider a cluster ID c and let the list N_c be $\{v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_k\}$. In the beginning, there is a triple $(v_1, c, size)$ in the priority queue with the current size of the cluster c due to the way the priority queue is initialized. When we process v_i, we add a triple with the updated size of c into the priority queue for the pair (v_{i+1}, c) in M_{v_i}. Hence, when we process v_{i+1} the size of the cluster will also be up-to-date (since it is the next adjacent vertex to cluster c being processed). When v is processed the triples with key value v are on top of the priority queue, due to the increasing processing order. □

Lemma 5.5

Processing a bucket and maintaining cluster sizes costs Sort($|\mathcal{T}|$) I/O-operations.

Proof. We need Sort($|\mathcal{T}|$) I/O-operations to sort the bucket and to build the lists N and $\mathcal{M}$. The operations `pop` and `push` of the priority queue have amortized $1/B \log_{M/B} N/B$ cost. The number of `push` (or `pop`) operations is equal to $|\mathcal{M}|$. This is due to the fact that each element of this list means that two vertices in the bucket have to take the size of the same cluster into account. Thus, we need to forward the information from the first to the second vertex. This means that the total cost of all operations is Sort($|\mathcal{T}|$). Iterating through the bucket costs Scan($|\mathcal{T}|$). Hence, processing a bucket costs Sort($|\mathcal{T}|$) in total. □

Theorem 5.6

One iteration of the coloring-based clustering algorithm costs Sort($|E|$) + $|C| \cdot$ Scan($|V|$) operations, where $|C|$ is the number of buckets.

Proof. We have $|C|$ buckets and processing each bucket costs Sort($|\mathcal{T}|$) I/Os. The overall number of elements in the buckets is $\mathcal{O}(|E|)$. Hence, processing all buckets takes Sort($|E|$) I/O-operations. After we processed a bucket, we update the sizes of the clusters that changed during processing. This takes Scan($|V|$) I/O-operations. Thus, one iteration of the bucket clustering algorithm costs Sort($|E|$) + $|C| \cdot$ Scan($|V|$) I/O-operations. □

5.2.3 Coarsening/Contraction

We present a (semi-)external algorithm to create graph hierarchies. In general, to create a graph hierarchy, we compute a size-constrained clustering of the current graph using one of the clustering algorithms described before. The next step is to renumber the cluster IDs. We sort the vertices by their cluster ID and scans the sorted array assigning new cluster IDs from $0, \ldots, n' - 1$, where n' is the number of the distinct clusters. This step can be done using $\text{Sort}(|V|)$ I/O operations. In contrast, the semi-external algorithm uses an additional array of size $\mathcal{O}(|V|)$ to assign new cluster IDs. Hence, it needs $\mathcal{O}(1)$ I/O operations.

External Algorithm. To compute the contracted graph, our external algorithm builds an array of triples $(\text{cluster}[u], \text{cluster}[v], w(u, v))$ for each edge $(u, v) \in E$. This array is sorted lexicographically by the first two entries using $\text{Sort}(|E|)$ I/Os. Then we merge parallel edges and build the edges of the quotient graph by iterating through the sorted array using $\text{Scan}(|E|)$ I/Os. The total I/O volume of this step is $\text{Sort}(|E|)$. The semi-external algorithm stores pairs $(\text{cluster}[u], \text{cluster}[v])$ for each edge (u, v) in a hash table and uses it to build the contracted graph. This can be done using $\text{Scan}(|E|)$ I/Os. If the number of edges of the contracted graphs decreases geometrically and a constant number of label propagation iterations is assumed, the complete hierarchy can be built using $\text{Scan}(|E|)$ I/Os using the semi-external algorithm or $\text{Sort}(|E|)$ I/Os using the external algorithm.

5.2.4 Uncoarsening/Projection of Partition

In this step, we want to project a partition of a coarse level to the next finer level in the hierarchy and perform some local search. Let $\mathcal{Q} = (V_\mathcal{Q}, E_\mathcal{Q})$ be a contracted graph of the next finer level $G = (V, E)$ and let $B_\mathcal{Q}$ be a partition of the contracted graph. Namely, $B_\mathcal{Q}[v]$ is the block ID of the vertex $v \in V_\mathcal{Q}$. Analogously, we denote the partition of G. Recall that the contracted graph has been built according to a clustering of the graph G. Note that a cluster i of G corresponds to a vertex i in $\mathcal{Q}$. Hence, for a vertex $v \in V$ the projected partition ID from the coarse level is $B_G[v] := B_\mathcal{Q}[\text{cluster}_G[v]]$, where cluster_G is the clustering of G according to which it was contracted.

To project the partition in our external algorithm, we build an array of pairs $(\text{cluster}_G[v], v)$ and sort it by the first component using $\text{Sort}(|V|)$ I/O operations. Now we iterate through the arrays $B_\mathcal{Q}$ and $\{(\text{cluster}_G[v], v)\}$ simultaneously and generate an array $\{(B_\mathcal{Q}[\text{cluster}_G[v]], v)\}$ which contains the projected solution. We sort the resulting array by the second component and apply the clustering to our graph. Overall, we need $\text{Sort}(|V|)$ I/O operations. The semi-external algorithm iterates through all vertices of graph G and updates the cluster IDs of each vertex. This can be done using $\mathcal{O}(1)$ I/Os. If the number of vertices in the quotient graphs decreases

geometrically then uncoarsening of the complete hierarchy can be done using Sort($|V|$) I/O operations. After each projection step, we apply the size-constrained LPA (using $L_{\max}$ as the size constraint) to improve the solution in on the current level.

5.3 Experimental Evaluation

In this section, we evaluate performance of our (semi-)external graph clustering and multi-level graph partitioning algorithms. We compare ourselves against `kMetis`, which is probably the most widely used partitioning algorithm, and `KaHIP` (see Section 3.8 for details).

Methodology. We implement our algorithms using `C++`. Our implementation uses the `STXXL 1.4.0` library [Bec+] to a large extent, i.e., external arrays, sorting algorithms and priority queues. All binaries were built using `g++-7.3.0`. In order to save time, we run our algorithm once reporting cut size, running time, internal memory consumption, and I/O volume. Experiments were run on Machine C (see Section 2.3.4 for details) using 3 SSD disks (read 1440 MB/s, write 1440 MB/s). The size of a block during the experiments is set to 1 MB. Using a larger block size does not yield an advantage due to parallel prefetching used in read/write algorithms that are implemented within the `STXXL` library.

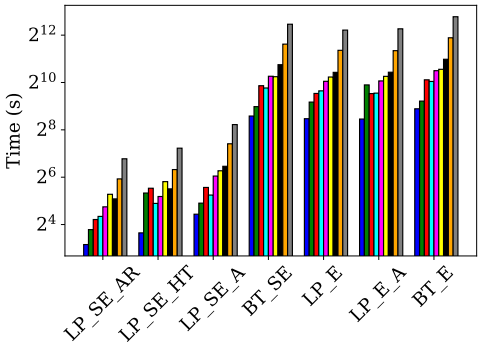
5.3.1 Graph Clustering Algorithms

We now evaluate different graph clustering algorithms with and without size constraint on nine large graphs from our collection (see Section 2.3.2). We use $L_{\max} := (1+\epsilon)\lceil c(V)/k \rceil$ with $k = 16$ as size constraint. We use the following algorithm abbreviations: LP – label propagation, SE – semi-external, E – external, BT – the coloring-based graph clustering algorithm that uses buckets, A – active vertex strategy, AR – using an array in label propagation, HT – using a hash table in label propagation, and SC – using an array in label propagation to maintain up-to-date cluster sizes. Here an array of size $|V|$ and a hash table are used to store connectivity information to adjacent clusters during label propagation. LP_SE_AR is the semi-external label propagation algorithm (see Section 5.2.1) that uses an array of size $|V|$ to calculate connectivity to adjacent clusters and LP_SE_HT uses a hash table. Note that label propagation with an array requires more memory than label propagation with a hash table but runs faster. LP_SE_A is the semi-external label propagation algorithm with the active vertex strategy (see Section 5.2.1). BT_SE is is the semi-external coloring-based clustering algorithm (see Section 5.2.2). Recall that we are allowed to have arrays of size $\mathcal{O}(|V|)$ in the internal memory and, thus, some information can be removed from tuples in buckets. LP_E is is the external label propagation algorithm (see Section 5.2.1). LP_E_SC is the external label propagation algorithm

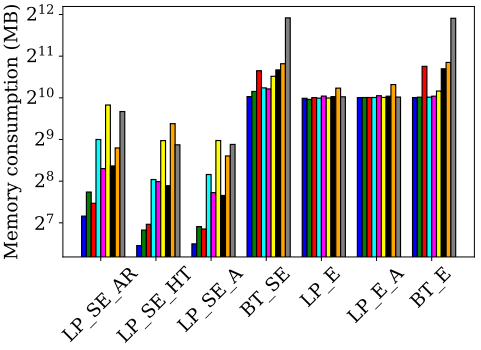
(see Section 5.2.1) that allows only one array of size $|V|$ in the internal memory to maintain up-to-date cluster sizes. BT_E_SC is the external coloring-based clustering algorithm (see Section 5.2.2). All algorithms perform three label propagation iterations. We use the variant of the coloring-based clustering algorithm which assumes that each bucket fits into the internal memory (this turned out to be true for all instances). In order to consume approximately at most 1GB of internal memory, the external priority queues in the external label propagation algorithm are allowed to use at most 170 MB of the internal memory each.

Figure 5.4 summarizes the results. First of all, both types of experiments (with or without a size constraint) show about the same results and further we analyze only the experiments without size constraint. Table 5.1 presents geometric mean values of running time, memory consumption and I/O volume. The semi-external label propagation algorithms outperform the semi-external coloring-based clustering algorithm. For example, LP_SE_AR runs in 109 s consuming 854 MB of the internal memory and its I/O volume is 161 GB on the graph uk-2007, whereas BT_SE runs in 5639 s consuming 4061 MB of the internal memory and its I/O volume is 743 GB on the same graph. Namely, LP_SE_AR is about 52.2 times faster, consumes about 4.8 times less of the internal memory and its I/O volume is about 4.6 times lower. This can be explained by the fact that each tuple in the buckets, which are used in BT_E, consists of three elements and has total size of 12 bytes (4 bytes per element of the tuple). Thus, sorting and scanning operations require a significantly larger amount of time, whereas the semi-external label propagation requires only scanning operations. However, the coloring-based clustering algorithm is able to compute a graph clustering fulfilling a size constraint in the external memory model.

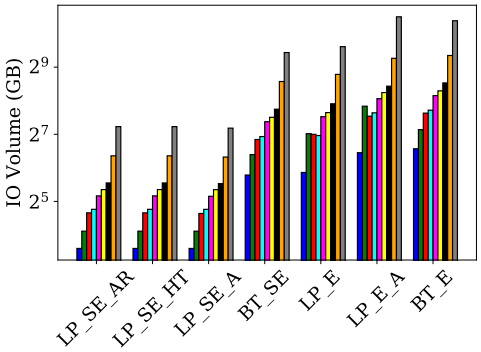
The external label propagation algorithms show comparable of better running times and I/O volumes than the external coloring-based clustering algorithm. For example, LP_E runs in 4748 s consuming 1 GB of the internal memory and its I/O volume is 839.4 GB on the graph uk-2007, whereas BT_E runs in 7023 s consuming 4 GB of the internal memory and its I/O volume is 1.4 TB on the same graph. LP_E is about 1.48 times faster, consumes about 4 times less of the internal memory and its I/O volume is about 1.7 times lower. This can be explained by the fact that each tuple in the buckets, which are used in BT_E, consists of five elements and has total size of 20 bytes (4 bytes per element of the tuple). The reduction of the gap between running times of both algorithms can be explained by the fact that label propagation algorithms use external priority queue whereas coloring-based algorithms use only external arrays. Table 5.2 presents measurements of all aforementioned clustering algorithms.



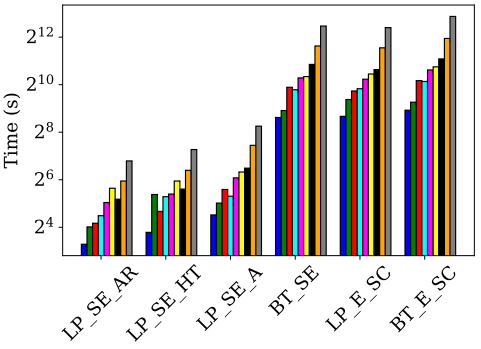
(a) Running times of clustering algorithms *without* a size constraint.



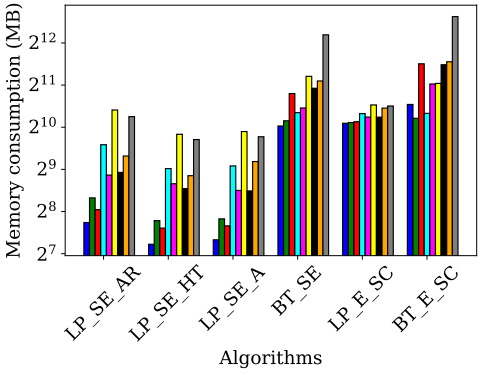
(b) Memory consumptions of clustering algorithms *without* a size constraint.



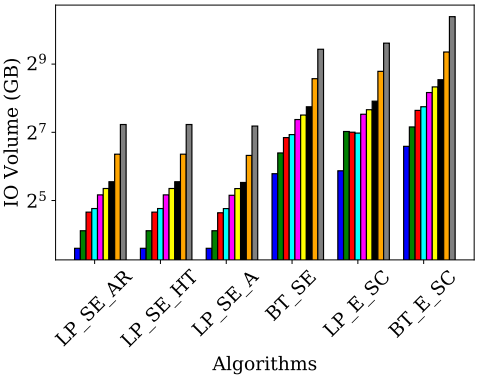
(c) I/O volumes of clustering algorithms *without* a size constraint.



(d) Running times of clustering algorithms *with* a size constraint.



(e) Memory consumptions of clustering algorithms *with* a size constraint.



(f) I/O volumes of clustering algorithms *with* a size constraint.

Figure 5.4: Evaluation of different clustering algorithms.

Table 5.1: Geometric mean of running time, memory consumption and I/O volume for clustering algorithms without a size constraint.

Algorithm	Time (s)	Memory consumption (MB)	I/O Volume (GB)
Without a size constraint			
LP_SE_AR	**28**	357	36.7
LP_SE_HT	45	244	36.7
LP_SE_A	66	**223**	**36.4**
BT_SE	1245	1529	168.8
LP_E	**1115**	**1045**	**192.8**
LP_E_A	1176	1061	297.9
BT_E	1500	1442	293.2
With a size constraint			
LP_SE_AR	**30**	530	36.7
LP_SE_HT	46	**382**	36.7
LP_SE_A	69	398	**36.4**
BT_SE	1268	1784	168.8
LP_E_SC	**1277**	**1253**	**193.7**
BT_E_SC	1595	2266	296.9

Table 5.2: Running time, memory consumption and I/O volume of different clustering algorithms.

Graph	Algorithm	Time (s)	Memory consumption (MB)	I/O Volume (GB)
Without a size constraint				
uk-2002	LP_SE_HT	12	**87**	**12.1**
uk-2002	LP_SE_AR	**8**	143	**12.1**
uk-2002	LP_SE_A	21	90	**12.1**
uk-2002	BT_SE	383	1045	55.2
uk-2002	LP_E	354	**1016**	**58.2**
uk-2002	LP_E_A	**351**	1028	87.5
uk-2002	BT_E	474	1028	94.9
nlpkkt240	LP_SE_HT	40	**113**	**17.3**
nlpkkt240	LP_SE_AR	**13**	213	**17.3**
nlpkkt240	LP_SE_A	29	120	**17.3**
nlpkkt240	BT_SE	505	1138	84.1

nlpkkt240	LP_E	**577**	**997**	**129.6**
nlpkkt240	LP_E_A	953	1029	228.6
nlpkkt240	BT_E	593	1035	140.9
arabic-2005	LP_SE_HT	46	124	25.3
arabic-2005	LP_SE_AR	**18**	177	25.3
arabic-2005	LP_SE_A	47	**115**	**24.9**
arabic-2005	BT_SE	931	1607	115.0
arabic-2005	LP_E	**742**	**1028**	**127.8**
arabic-2005	LP_E_A	740	1029	186.0
arabic-2005	BT_E	1107	1729	198.2
rgg_2_26	LP_SE_HT	29	**263**	**27.2**
rgg_2_26	LP_SE_AR	**20**	512	**27.2**
rgg_2_26	LP_SE_A	37	286	**27.2**
rgg_2_26	BT_SE	871	1208	122.2
rgg_2_26	LP_E	801	**1018**	**124.7**
rgg_2_26	LP_E_A	**749**	1029	199.4
rgg_2_26	BT_E	1055	1035	210.4
uk-2005	LP_SE_HT	36	254	35.9
uk-2005	LP_SE_AR	**26**	315	35.9
uk-2005	LP_SE_A	66	**211**	**35.6**
uk-2005	BT_SE	1228	1187	165.9
uk-2005	LP_E	**1060**	1053	**184.1**
uk-2005	LP_E_A	1072	1060	267.1
uk-2005	BT_E	1447	**1052**	284.3
webbase-2001	LP_SE_HT	56	**502**	40.9
webbase-2001	LP_SE_AR	**38**	908	40.9
webbase-2001	LP_SE_A	77	503	**40.8**
webbase-2001	BT_SE	1213	1464	181.8
webbase-2001	LP_E	**1202**	**1021**	**200.4**
webbase-2001	LP_E_A	1225	1030	303.0
webbase-2001	BT_E	1506	1146	313.9
it-2004	LP_SE_HT	45	237	46.9
it-2004	LP_SE_AR	**33**	329	46.9
it-2004	LP_SE_A	88	**201**	**46.2**
it-2004	BT_SE	1724	1629	215.0
it-2004	LP_E	**1377**	**1045**	**240.2**
it-2004	LP_E_A	1383	1052	345.8
it-2004	BT_E	2021	1660	369.7

sk-2005	LP_SE_HT	80	666	82.0
sk-2005	LP_SE_AR	**60**	444	82.0
sk-2005	LP_SE_A	170	**389**	**80.0**
sk-2005	BT_SE	3148	1805	380.7
sk-2005	LP_E	2626	**1203**	**441.0**
sk-2005	LP_E_A	**2600**	1277	614.5
sk-2005	BT_E	3791	1844	651.5
uk-2007	LP_SE_HT	149	**468**	150.0
uk-2007	LP_SE_AR	**109**	815	150.0
uk-2007	LP_SE_A	298	471	**145.3**
uk-2007	BT_SE	5639	3873	692.3
uk-2007	LP_E	**4748**	1041	**781.8**
uk-2007	LP_E_A	4929	**1037**	1448.5
uk-2007	BT_E	7023	3846	1334.7

With a size constraint

Graph	Algorithm	Time (s)	Memory consumption (MB)	I/O Volume (GB)
uk-2002	LP_SE_HT	13	**149**	**12.1**
uk-2002	LP_SE_AR	**9**	213	**12.1**
uk-2002	LP_SE_A	22	160	**12.1**
uk-2002	BT_SE	393	1045	55.2
uk-2002	LP_E_SC	**406**	**1092**	**58.5**
uk-2002	BT_E_SC	486	1489	96.3
nlpkkt240	LP_SE_HT	41	**220**	**17.3**
nlpkkt240	LP_SE_AR	**16**	320	**17.3**
nlpkkt240	LP_SE_A	32	226	**17.3**
nlpkkt240	BT_SE	482	1138	84.1
nlpkkt240	LP_E_SC	664	**1105**	**130.1**
nlpkkt240	BT_E_SC	**614**	1186	142.7
arabic-2005	LP_SE_HT	25	**194**	25.3
arabic-2005	LP_SE_AR	**18**	263	25.3
arabic-2005	LP_SE_A	48	202	**24.9**
arabic-2005	BT_SE	949	1780	115.0
arabic-2005	LP_E_SC	**851**	**1121**	**128.3**
arabic-2005	BT_E_SC	1152	2907	199.9
rgg_2_26	LP_SE_HT	39	**519**	**27.2**
rgg_2_26	LP_SE_AR	**22**	768	**27.2**
rgg_2_26	LP_SE_A	39	542	**27.2**

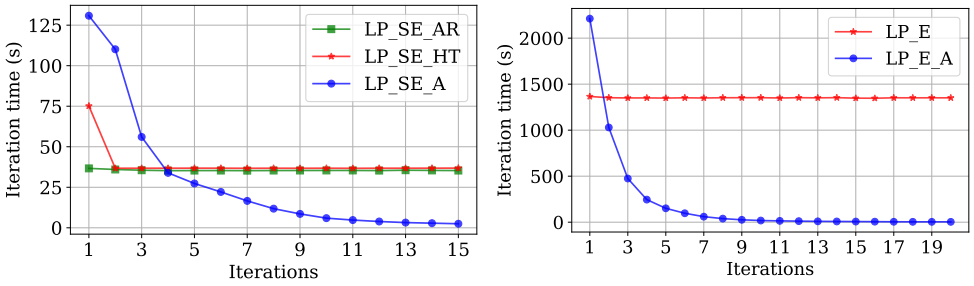
rgg_2_26	BT_SE	882	1302	122.2
rgg_2_26	LP_E_SC	**909**	**1280**	**126.0**
rgg_2_26	BT_E_SC	1123	1286	215.2
uk-2005	LP_SE_HT	42	404	35.9
uk-2005	LP_SE_AR	**32**	466	35.9
uk-2005	LP_SE_A	67	**362**	**35.6**
uk-2005	BT_SE	1247	1406	165.9
uk-2005	LP_E_SC	**1201**	**1210**	**184.8**
uk-2005	BT_E_SC	1572	2085	287.2
webbase-2001	LP_SE_HT	61	**912**	40.9
webbase-2001	LP_SE_AR	**49**	1359	40.9
webbase-2001	LP_SE_A	80	953	**40.8**
webbase-2001	BT_SE	1296	2366	181.8
webbase-2001	LP_E_SC	**1398**	**1477**	**202.6**
webbase-2001	BT_E_SC	1721	2106	322.1
it-2004	LP_SE_HT	48	373	46.9
it-2004	LP_SE_AR	**36**	487	46.9
it-2004	LP_SE_A	89	**359**	**46.2**
it-2004	BT_SE	1849	1944	215.0
it-2004	LP_E_SC	**1592**	**1209**	**241.0**
it-2004	BT_E_SC	2170	2863	372.6
sk-2005	LP_SE_HT	84	**461**	82.0
sk-2005	LP_SE_AR	**61**	638	82.0
sk-2005	LP_SE_A	174	582	**80.0**
sk-2005	BT_SE	3165	2192	380.7
sk-2005	LP_E_SC	**2998**	**1402**	**441.9**
sk-2005	BT_E_SC	3943	3005	654.8
uk-2007	LP_SE_HT	154	**836**	150.0
uk-2007	LP_SE_AR	**110**	1219	150.0
uk-2007	LP_SE_A	305	875	**145.3**
uk-2007	BT_SE	5658	4681	692.3
uk-2007	LP_E_SC	**5411**	**1451**	**783.8**
uk-2007	BT_E_SC	7490	6321	1342.5

Active Vertex Strategy. Additionally, we evaluate label propagation with and without the active vertex strategy for more iterations; our experiments indicate that this strategy decreases the running time of label propagation in both memory models if the number of iterations is sufficiently large. Specifically, we run LP_SE_AR, LP_SE_HT, and LP_SE_A for 15 iterations and LP_E and LP_E_A for 20 iterations. Table 5.3 shows that the active vertex strategy speed ups computations. More specifically, the geometrical mean running times of LP_SE_A and LP_E_A are 133 s and 1867 s, respectively. The geometrical mean running times of LP_SE_AR and LP_E are 141 s and 6527 s, respectively. Thus, on average LP_SE_A and LP_E_A are 1.1 and 3.5 times faster than LP_SE_AR and LP_E, respectively. In the most extreme case, LP_SE_A is a factor of 1.6 faster than LP_SE_AR on the graph uk-2007 and LP_E_A is a factor of 5.1 faster than than LP_E on the graph sk-2005.

Furthermore, Figures 5.5 (a) and 5.5 (b) show running times of every iteration of (semi-)external algorithms on the graph uk-2007. Specifically, the fourth iteration of LP_SE_A is already faster than those of LP_SE_AR and LP_SE_HT. Further, the second iteration of LP_E_A is already faster than that of LP_E. This is not surprising since in the external algorithms we store vertices in external priority queues and, thus, the less vertices are in an external priority queue the faster is an iteration.

Table 5.3: Running times in seconds of different (semi-)external LPAs.

Graph	Semi-external LPAs (15 iterations)			External LPAs (20 iterations)	
	LP_SE_AR	LP_SE_HT	LP_SE_A	LP_E	LP_E_A
uk-2002	43	47	**42**	2040	**474**
nlpkkt240	**64**	159	143	**3543**	5871
arabic-2005	91	119	**81**	4310	**918**
rgg_2_26	102	111	**72**	4733	**967**
uk-2005	131	141	**115**	6136	**1373**
webbase-2001	**179**	220	197	6840	**2726**
it-2004	173	185	**151**	8074	**1724**
sk-2005	305	324	**253**	15496	**3015**
uk-2007	709	590	**440**	27753	**5719**
Geometrical mean	141	168	**133**	6527	**1867**



(a) Semi-external algorithms. **(b)** External algorithms.

Figure 5.5: Running times of iterations of different (semi-)external LPAs on the graph uk-2007.

5.3.2 Multi-level Graph Partitioning

We now present the results of the semi-external multi-level graph partitioning algorithm. To save running time, we evaluate only graph partitioning algorithms that use the semi-external LPA with a size constraint or its parallel version in the coarsening and uncoarsening phases. We evaluate both versions of parallel LPAs (see Section 5.2.1) since P_LP_SE_AR is the faster one and P_LP_SE_HT requires *less* internal memory (here P stands for parallel). Specifically, P_LP_SE_AR is the semi-external label propagation algorithm that uses an array of size $|V|$ to calculate connectivity to adjacent clusters and P_LP_SE_HT uses a hash table. Note that P_LP_SE_AR requires more memory than P_LP_SE_HT. This is especially important when the number of PEs is large: since each PE uses an array of size $|V|$ in P_LP_SE_AR and, thus, the required amount of internal memory is $\mathcal{O}(p|V|)$ instead of $\mathcal{O}(p\Delta)$ required by P_LP_SE_HT. Our experiments focus on 13 large graphs from our benchmark set (see Section 2.3.2). Since we partition a small number of the graphs, we do not perform statistical significant tests. During coarsening, we use $L_{\max}$ as a size constraint. Note that this is a much weaker restriction on the cluster sizes than the one used by Meyerhenke et al. [MSS14]. Using this weaker constraint speeds up the algorithm significantly. We partition all graphs using $k = 16$ and $\epsilon = 0.03$. This is one of the values used in [WC00a] and `kMetis`. Furthermore, we remove high degree vertices from the coarse graph before initial partitioning when partition twitter-2010, clueweb-12, uk-2014, and eu-2015 since otherwise initial partitioning using `KaHIP` runs too long. Tables 5.4 and 5.5 summarize the results.

Comparison of Semi-external Memory Algorithms. LP_SE_AR produces smaller cuts than other algorithms on average. Specifically, LP_SE_AR cuts 6.2%, 0.3%, and 5.8% less edges than LP_SE_HT, P_LP_SE_AR, and P_LP_SE_HT, respectively. LP_SE_HT and P_LP_SE_HT have worse average quality than

LP_SE_AR and P_LP_SE_AR since they partition the graph clueweb12 with the cut sizes a factor of 2 larger than those of LP_SE_AR and P_LP_SE_AR. The reason is that during initial partitioning in LP_SE_HT and P_LP_SE_HT `KaHIP` produces large cuts. This can be explained that by the fact that during the coarsening phase a part of the graph with a small cut was contracted into one vertex. Choosing another random seed should solve the problem. The same reasoning explains poor quality of LP_SE_AR and LP_SE_HT on the graph rgg_2_26. Note that on the other graphs all algorithms have comparable quality. Additionally, Figure 5.6 shows the performance plots (see Section 2.3.1) that compares our semi-external memory algorithms. We can see that the parallel algorithms produces partitions of slightly worse quality than the sequential algorithms. This is not surprising since the parallel label propagation algorithm applies moves of vertices between clusters not immediately. Thus, the clustering constructed by a parallel algorithm may be worse for further partitioning. However, we expect that additional iterations of label propagation should solve the problem. This explains poor quality of P_LP_SE_AR and P_LP_SE_HT on the graph sk-2005. In summary, we can see that parallelization only slightly worsens partition quality and, thus, yields a good trade-off between quality and running time.

P_LP_SE_AR runs faster than other algorithms on average. P_LP_SE_AR is 1.08, 1.02, and 1.04 faster than LP_SE_AR, LP_SE_HT, and P_LP_SE_HT, respectively. These small speed ups can be explained by the fact that reading from disks is a bottleneck. Furthermore, initial partitioning is a bottleneck on some instances. But if we consider large graphs, we can see that both P_LP_SE_AR and P_LP_SE_HT perform faster than their corresponding sequential versions. For example, on the graph eu-2015 P_LP_SE_AR is 1.6 times faster than LP_SE_AR and P_LP_SE_HT is 1.3 is faster than LP_SE_AR. Note that on some graphs P_LP_SE_AR is slower than LP_SE_AR since the coarsest graph computed by P_LP_SE_AR is not small enough or has high degree vertices and `KaHIP` in initial partitioning runs longer than on the coarsest graph computed by LP_SE_AR.

LP_SE_HT consumes less memory than other algorithms on average. Specifically, LP_SE_HT consumes 1.1, 2.1, and 1.8 times less memory than LP_SE_AR, P_LP_SE_AR, and P_LP_SE_HT, respectively. If we consider large graphs, we can see that P_LP_SE_HT consumes significantly less memory than P_LP_SE_AR. For example, on the graph eu-2015 P_LP_SE_AR consumes about 78 GB wheres P_LP_SE_HT consumes only 16 GB.

Comparison to Competitors. All our algorithm show comparable quality and performance. Therefore, we analyze only the results of P_LP_SE_AR and the competitors.

P_LP_SE_AR computes partitions that are almost as good as those computed by `KaHIP`. On average, we cut about 5% more edges. Furthermore, in the worst case, our algorithm cuts about 15% more edges than `KaHIP` on the graph arabic-2005. In contrast, our algorithm computes much better cuts than `kMetis` (except on the graph

nlpkkt240). On average, our algorithm cuts 31% less edges than `kMetis`. Moreover, in the best case, our algorithm cuts 49% less edges than `kMetis` on the graph it-2004. Additionally, Figure 5.7 shows the performance plots for our semi-external memory algorithms and our competitors. We can see that our algorithms produce partitions of quality comparable to that of `KaHIP` and it is always better than `kMetis`.

Furthermore, P_LP_SE_AR is faster than `KaHIP` and almost always faster than `kMetis` (except on the graph rgg_2_26). On average, we are 2.9 and 1.1 times faster than `KaHIP` and `Metis`, respectively. Furthermore, in the best case, we are a factor of 5.3 faster than `KaHIP` on the graph rgg_2_26 and a factor of 1.7 faster than `Metis` on the graph it-2004. This is partially due to the fact that we use a weaker size constraint during coarsening which makes the contracted graph even smaller than the contracted graphs computed by KaHIP and the fact that we use less label propagation rounds. After the first contraction step, we switch to the internal memory implementation of `KaHIP` to partition the coarser graph.

In terms of memory consumption, P_LP_SE_AR always consumes less memory than competitors. On average, P_LP_SE_AR consumes 9.8 and 6.7 times less memory than `KaHIP` and `Metis`, respectively. Furthermore, in the best case, we consume a factor of 11.3 and a factor of 9.1 less memory than `KaHIP` and `Metis`, respectively, on the graph it-2004.

Table 5.4: Geometrical means of running time and memory consumption of different partitioning algorithms. P_LP_SE_HT and P_LP_SE_AR use 15 PEs (without hyperthreading) and the column "Memory consumption (MB)" shows the amount of the internal memory used by the algorithms in megabytes.

Algorithm	Total Time (s)	LP Time (s)	Cut	Memory consumption (MB)
Instances where all algorithms succeeded				
KaHIP	254	–	**2.4M**	21893
LP_SE_AR	**85**	34	2.7M	1068
LP_SE_HT	91	45	2.8M	**1017**
Metis	99	–	3.6M	14977
P_LP_SE_AR	88	**29**	2.5M	2237
P_LP_SE_HT	101	42	2.5M	2003
All instances				
LP_SE_AR	444	140	**13.9M**	3209
LP_SE_HT	420	162	14.8M	**3032**
P_LP_SE_AR	**413**	**112**	13.9M	6258
P_LP_SE_HT	429	146	14.7M	5323

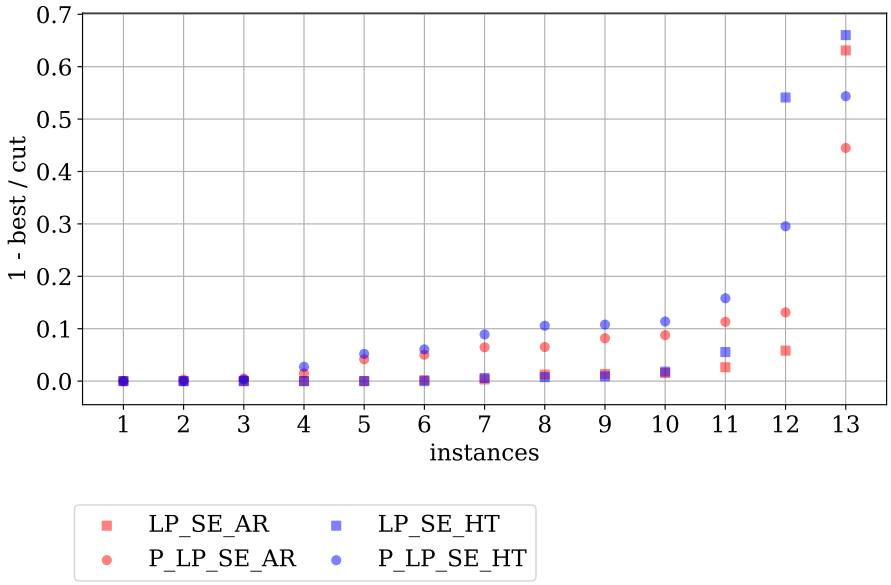


Figure 5.6: Performance plot for the cut size on thirteen graphs.

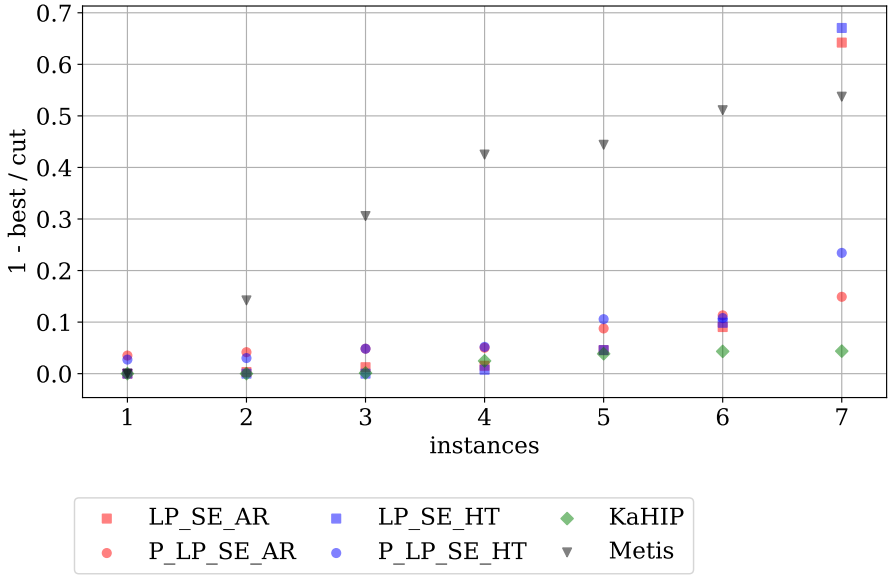


Figure 5.7: Performance plot for the cut size on seven graphs.

Table 5.5: Running time and memory consumption of different partitioning algorithms. P_LP_SE_HT and P_LP_SE_AR use 15 PEs (without hyper-threading) and the column "Memory consumption (MB)" shows the amount of the internal memory used by the algorithms in megabytes.

Graph	Algorithm	All Time (s)	LP Time (s)	Cut	Memory consumption (MB)
uk-2002	KaHIP	91	–	1.5M	9422
uk-2002	LP_SE_AR	50	16	1.5M	533
uk-2002	LP_SE_HT	47	18	**1.5M**	**526**
uk-2002	Metis	45	–	2.5M	6820
uk-2002	P_LP_SE_AR	**34**	**9**	1.5M	1386
uk-2002	P_LP_SE_HT	35	10	1.5M	1025
nlpkkt240	KaHIP	125	–	5.7M	16074
nlpkkt240	LP_SE_AR	53	25	5.7M	681
nlpkkt240	LP_SE_HT	86	59	5.7M	**675**
nlpkkt240	Metis	41	–	**5.5M**	11073
nlpkkt240	P_LP_SE_AR	37	**13**	5.7M	2086
nlpkkt240	P_LP_SE_HT	**36**	14	5.7M	681
arabic-2005	KaHIP	137	–	**1.9M**	18508
arabic-2005	LP_SE_AR	58	26	2.1M	460
arabic-2005	LP_SE_HT	55	30	2.1M	**454**
arabic-2005	Metis	67	–	3.4M	15173
arabic-2005	P_LP_SE_AR	46	20	2.2M	1672
arabic-2005	P_LP_SE_HT	**44**	**19**	2.5M	924
rgg_2_26	KaHIP	425	–	**218.6K**	24672
rgg_2_26	LP_SE_AR	**71**	34	610.9K	1033
rgg_2_26	LP_SE_HT	80	47	663.5K	**771**
rgg_2_26	Metis	58	–	254.8K	10467
rgg_2_26	P_LP_SE_AR	79	**23**	226.5K	5056
rgg_2_26	P_LP_SE_HT	72	26	225.4K	3318
uk-2005	KaHIP	374	–	**3.4M**	28154
uk-2005	LP_SE_AR	**109**	44	3.5M	1366
uk-2005	LP_SE_HT	**109**	50	3.4M	**1359**
uk-2005	Metis	174	–	7.0M	19822
uk-2005	P_LP_SE_AR	115	**27**	3.6M	2954
uk-2005	P_LP_SE_HT	128	37	3.5M	2579
webbase-2001	KaHIP	716	–	10.3M	35864
webbase-2001	LP_SE_AR	**197**	67	10.0M	4951
webbase-2001	LP_SE_HT	208	82	**9.9M**	**4939**
webbase-2001	Metis	465	–	14.2M	25419

webbase-2001	P_LP_SE_AR	344	**48**	11.1M	8940
webbase-2001	P_LP_SE_HT	345	56	11.1M	8265
it-2004	KaHIP	378	–	3.3M	34524
it-2004	LP_SE_AR	137	53	**3.2M**	1354
it-2004	LP_SE_HT	131	58	3.2M	**1348**
it-2004	Metis	154	–	6.8M	27967
it-2004	P_LP_SE_AR	**91**	**30**	3.5M	3059
it-2004	P_LP_SE_HT	103	45	3.5M	2834
twitter-2010	KaHIP	–	–	–	–
twitter-2010	LP_SE_AR	24943	542	673.1M	23180
twitter-2010	LP_SE_HT	13777	467	671.2M	**21299**
twitter-2010	Metis	–	–	–	–
twitter-2010	P_LP_SE_AR	**1549**	**163**	643.6M	30904
twitter-2010	P_LP_SE_HT	1651	183	**634.0M**	29369
sk-2005	KaHIP	–	–	–	–
sk-2005	LP_SE_AR	329	85	41.5M	1085
sk-2005	LP_SE_HT	**260**	94	**40.5M**	**1037**
sk-2005	Metis	–	–	–	–
sk-2005	P_LP_SE_AR	776	**52**	72.9M	3789
sk-2005	P_LP_SE_HT	355	84	57.4M	3122
uk-2007	KaHIP	–	–	–	–
uk-2007	LP_SE_AR	341	155	**4.1M**	4065
uk-2007	LP_SE_HT	304	166	4.1M	**4053**
uk-2007	Metis	–	–	–	–
uk-2007	P_LP_SE_AR	276	**92**	4.3M	7772
uk-2007	P_LP_SE_HT	**269**	100	4.3M	6054
clueweb12	LP_SE_AR	**8165**	2404	738.1M	98182
clueweb12	LP_SE_HT	8209	2315	1587.1M	**98143**
clueweb12	P_LP_SE_AR	21262	**1167**	**728.2M**	98248
clueweb12	P_LP_SE_HT	16465	1206	1595.6M	98251
uk-2014	LP_SE_AR	4067	1917	**43.7M**	13953
uk-2014	LP_SE_HT	3662	2119	44.5M	**13900**
uk-2014	P_LP_SE_AR	**2936**	**1182**	50.3M	57489
uk-2014	P_LP_SE_HT	3050	1315	49.3M	22661
eu-2015	LP_SE_AR	8489	4095	166.5M	17247
eu-2015	LP_SE_HT	7058	3959	**166.2M**	**13279**
eu-2015	P_LP_SE_AR	**5401**	**2188**	180.9M	78466
eu-2015	P_LP_SE_HT	5516	2195	182.4M	16621

Scalability of LPAs. Figures 5.8 – 5.11 present the scalability of P_LP_SE_AR and P_LP_SE_HT algorithms (left and right columns, respectively) on 13 large graphs. P_LP_SE_AR with 15 PEs achieves the best absolute speed-up of 3.3 on the graph twitter-2010. Furthermore, P_LP_SE_AR with 15 PEs achieves the best

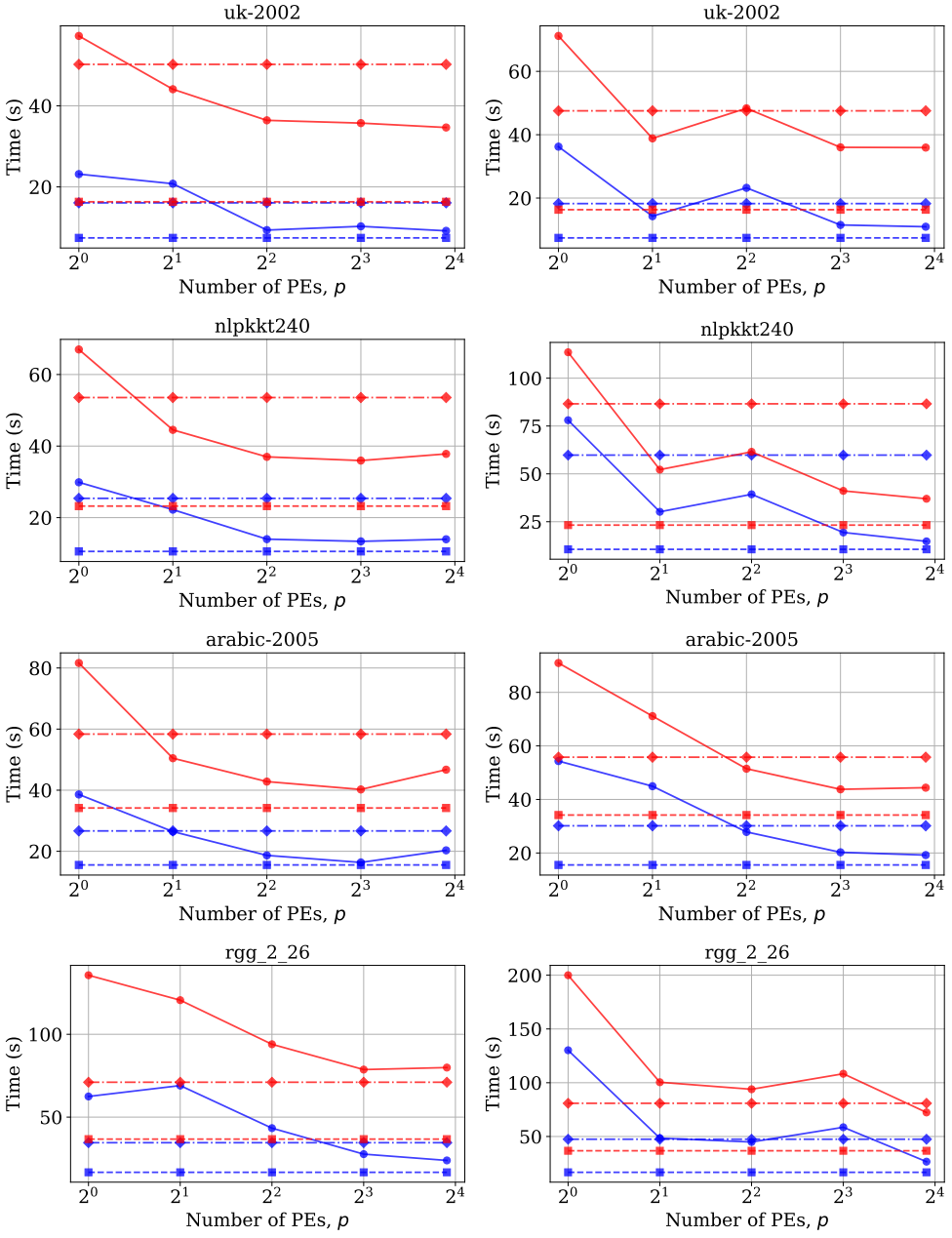
relative speed-up of 4.3 on the graph twitter-2010, whereas P_LP_SE_HT with 15 PEs achieves the best absolute and relative speed-up of 4.1 and 5.3 (respectively) on the graph nlpkkt240. Both algorithms only scale moderately because reading from disks is a bottleneck and parallel prefetching has a PE for each disk. Nevertheless, parallel algorithms with 15 PEs always speed up the computations compared to the sequential algorithms. Moreover, parallel algorithms get close to the lower bound which is given by the running time of a scan operation that only reads from disks the same I/O volume performed by the algorithms. P_LP_SE_HT scales better on most of the graphs than P_LP_SE_AR since in P_LP_SE_HT each PE uses a hash table instead of an array of size $|V|$ as in P_LP_SE_AR. Thus, each hash table fits into caches decreasing the effects of memory bandwidth. Another reasons why P_LP_SE_HT scales better: P_LP_SE_AR is already fast enough – due to the faster access to an array than to a hash table – and reading from disks remains the only bottleneck. For example, P_LP_SE_AR scales poorly after using more than two PEs on sk-2005 and uk-2007.

We emphasize again that speed-ups are small because reading from disks is a bottleneck. To see that, consider Figure 5.12 that presents speed-ups of average block processing running times (i.e., the running time to process a block without time to read it from a disk divided by the number of processed blocks) of the parallel LPAs for uk-2007. We can see that the algorithms scale better now. These harmonic mean speed-ups are not high since the memory bandwidth is a bottleneck, P_LP_SE_AR and P_LP_SE_HT use static load balancing and parallel prefetching has a PE for each disk. Furthermore, we expect that using blocks of size more than 1 MB may yield better speed-ups of block processing running time. However, we do not expect the overall running time of LPA to change since reading a block will take more time than its processing and, thus, the speed-up of block processing will not affect the overall running time. Table 5.6 presents the absolute and relative speed-ups of the parallel LPAs and corresponding block processing. P_LP_SE_HT has worse absolute speed-ups because the cluster IDs are updated only after processing a block which means that the LPA inserts more *different* keys into a hash table. Thus, P_LP_SE_HT has worse data locality than its sequential version.

Graph	Absolute speed-up of LP		Absolute speed-up of block processing	
	P_LP_SE_AR	P_LP_SE_HT	P_LP_SE_AR	P_LP_SE_HT
uk-2002	**1.8**	1.7	**3.1**	2.2
nlpkkt240	1.8	**4.1**	3.2	**4.3**
arabic-2005	1.3	**1.6**	**3.1**	2.6
rgg_2_26	1.4	**1.8**	**1.9**	1.8
uk-2005	**1.6**	1.3	**2.2**	2.0
webbase-2001	1.4	**1.5**	1.7	**1.8**
it-2004	**1.7**	1.3	**3.2**	2.2
twitter-2010	**3.3**	2.6	**3.4**	2.6
sk-2005	**1.6**	1.1	**3.1**	2.3
uk-2007	**1.7**	1.7	**3.3**	2.5
clueweb12	**2.1**	1.9	**3.2**	2.7
uk-2014	**1.6**	1.6	**3.6**	2.6
eu-2015	**1.9**	1.8	**4.4**	3.8
Harmonic mean	**1.7**	1.7	**2.9**	2.4

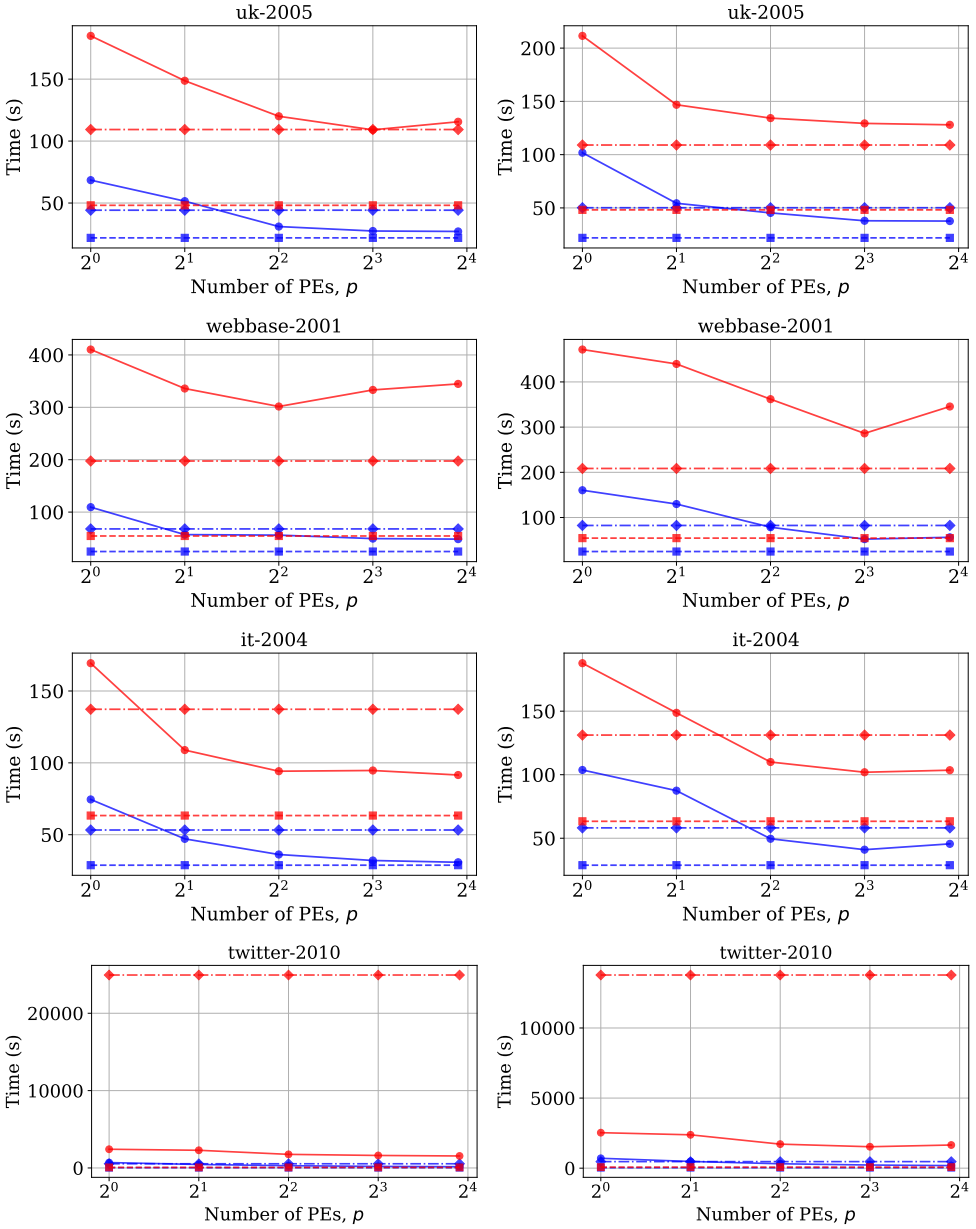
Graph	Relative speed-up of LP		Relative speed-up of block processing	
	P_LP_SE_AR	P_LP_SE_HT	P_LP_SE_AR	P_LP_SE_HT
uk-2002	2.5	**3.3**	**4.4**	4.3
nlpkkt240	2.1	**5.3**	3.7	**5.6**
arabic-2005	1.9	**2.8**	4.4	**4.8**
rgg_2_26	2.6	**4.9**	3.5	**5.0**
uk-2005	2.5	**2.7**	3.3	**4.1**
webbase-2001	2.3	**2.9**	2.8	**3.5**
it-2004	**2.4**	2.3	**4.5**	3.9
twitter-2010	**4.3**	3.8	**4.3**	3.9
sk-2005	**2.6**	2.1	**4.8**	4.4
uk-2007	2.5	**2.9**	**4.7**	4.4
clueweb12	2.9	**4.4**	4.5	**6.2**
uk-2014	2.5	**3.0**	**5.4**	4.9
eu-2015	2.7	**3.5**	6.3	**7.5**
Harmonic mean	2.5	**3.2**	4.2	**4.6**

Table 5.6: Absolute and relative speed-ups of P_LP_SE_AR and P_LP_SE_HT with 15 PEs.



(a) Running times of P_LP_SE_AR **(b)** Running times of P_LP_SE_HT

Figure 5.8: Running times of parallel LPAs.



(a) Running times of P_LP_SE_AR **(b)** Running times of P_LP_SE_HT

Figure 5.9: Running times of the parallel LPAs.

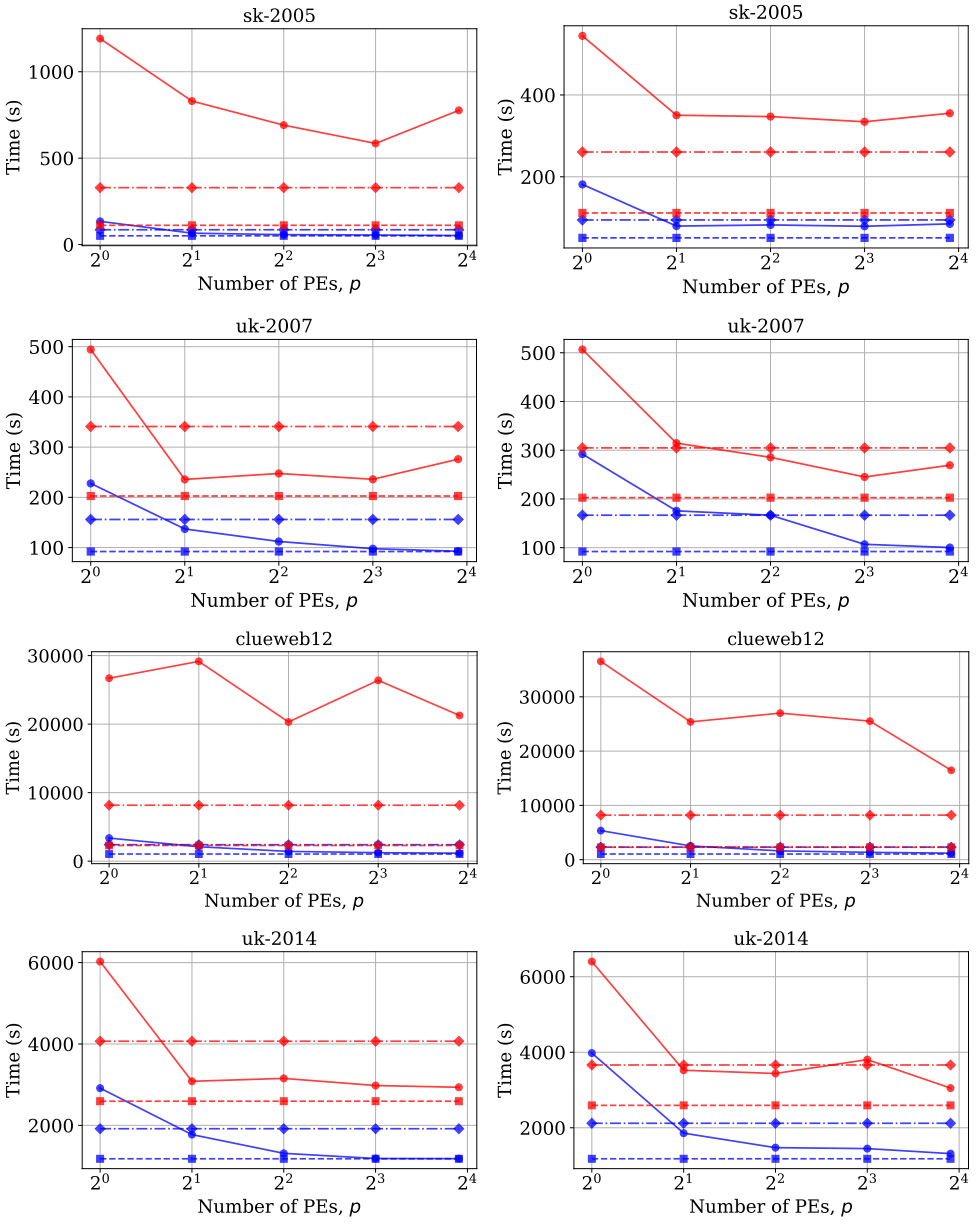
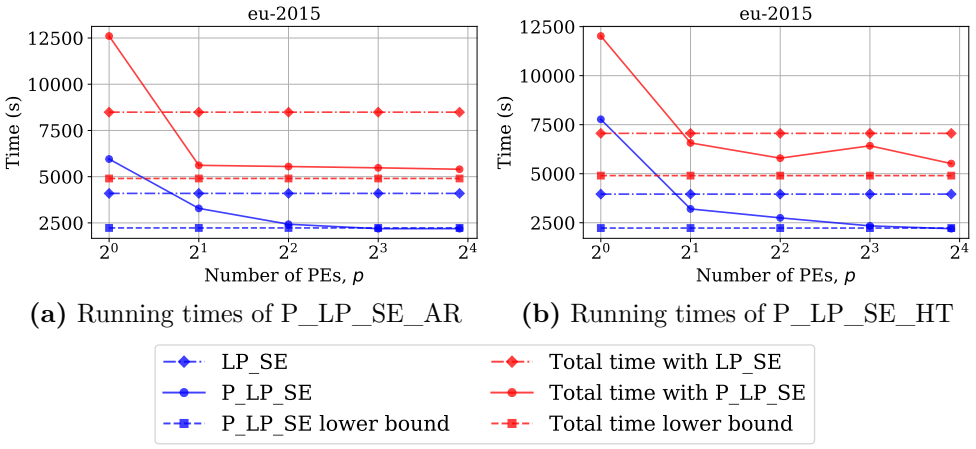
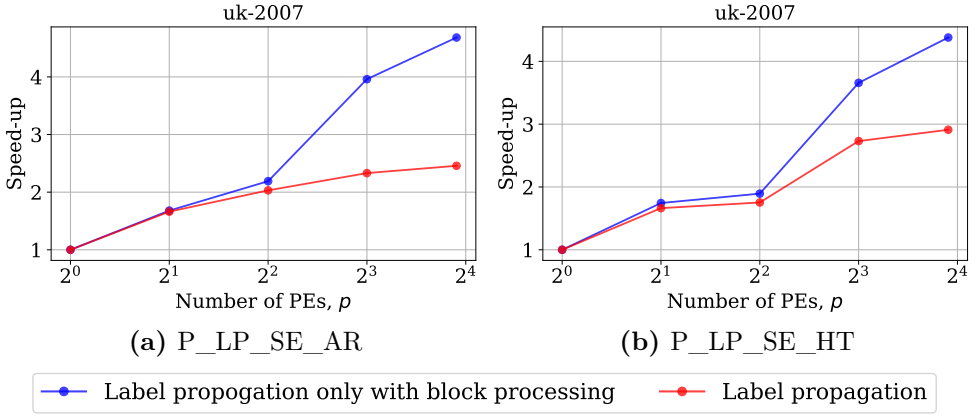


Figure 5.10: Running times of parallel LPAs.



(a) Running times of P_LP_SE_AR **(b)** Running times of P_LP_SE_HT

Figure 5.11: Running times of parallel LPAs.



(a) P_LP_SE_AR **(b)** P_LP_SE_HT

Figure 5.12: Speed-ups of parallel LPAs with and without time to read blocks on the graph uk-2007.

Memory Consumption of LPAs. Table 5.7 presents memory consumptions of the LPAs. First of all, LP_SE_HT and P_LP_SE_HT use significantly less memory than LP_SE_AR and P_LP_SE_AR. This is not surprising since LP_SE_HT and P_LP_SE_HT use hash tables during label propagation to find clusters with the strongest connection, whereas LP_SE_AR and P_LP_SE_AR use an array of size $|V|$. Figures 5.13, 5.14 present an additional comparison of memory consumptions for the parallel LPAs. Note that memory consumption of P_LP_SE_AR grows much

faster than that of P_LP_SE_HT with the number of PEs. For example, on the graph uk-2007 P_LP_SE_HT uses only 848 MB with one PE, whereas P_LP_SE_AR uses 1624 MB. Furthermore, P_LP_SE_HT uses only 1056 MB with 15 PEs, whereas P_LP_SE_AR uses 7280 MB on the same graph. Thus, P_LP_SE_HT uses 1.9 times less memory with one PE but with 15 PEs it uses 6.9 less memory.

Graph	LP_SE_AR	LP_SE_HT	P_LP_SE_AR	P_LP_SE_HT
uk-2002	220	**142**	1277	**175**
nlpkkt240	326	**213**	1928	**222**
arabic-2005	270	**187**	1568	**381**
rgg_2_26	774	**512**	4614	**521**
uk-2005	472	**396**	2716	**1639**
webbase-2001	1366	**904**	8119	**959**
it-2004	494	**365**	2842	**1021**
twitter-2010	513	**399**	2936	**1466**
sk-2005	645	**452**	3484	**1311**
uk-2007	1228	**824**	7280	**1056**
clueweb12	11704	**10810**	67221	**53974**
uk-2014	9117	**6173**	54137	**8314**
eu-2015	12445	**8272**	73586	**9696**
Geometrical mean	1076	**767**	6251	**1434**

Table 5.7: Memory consumption in megabytes of LP_SE_AR, LP_SE_HT, P_LP_SE_AR and P_LP_SE_HT with 15 PEs.

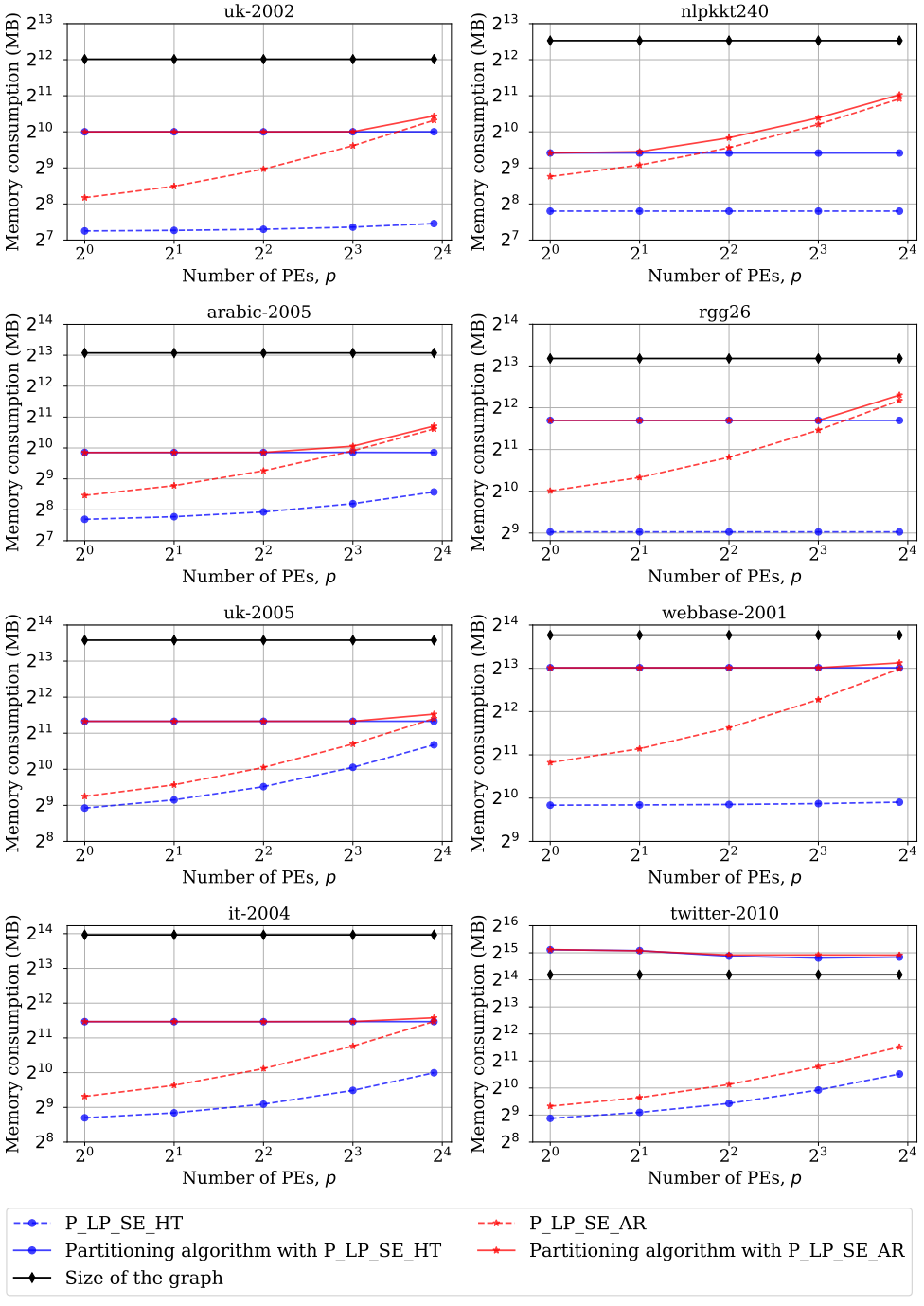


Figure 5.13: Memory consumption of our parallel LPAs. Here the black curve is the memory consumption of the graph in the internal memory.

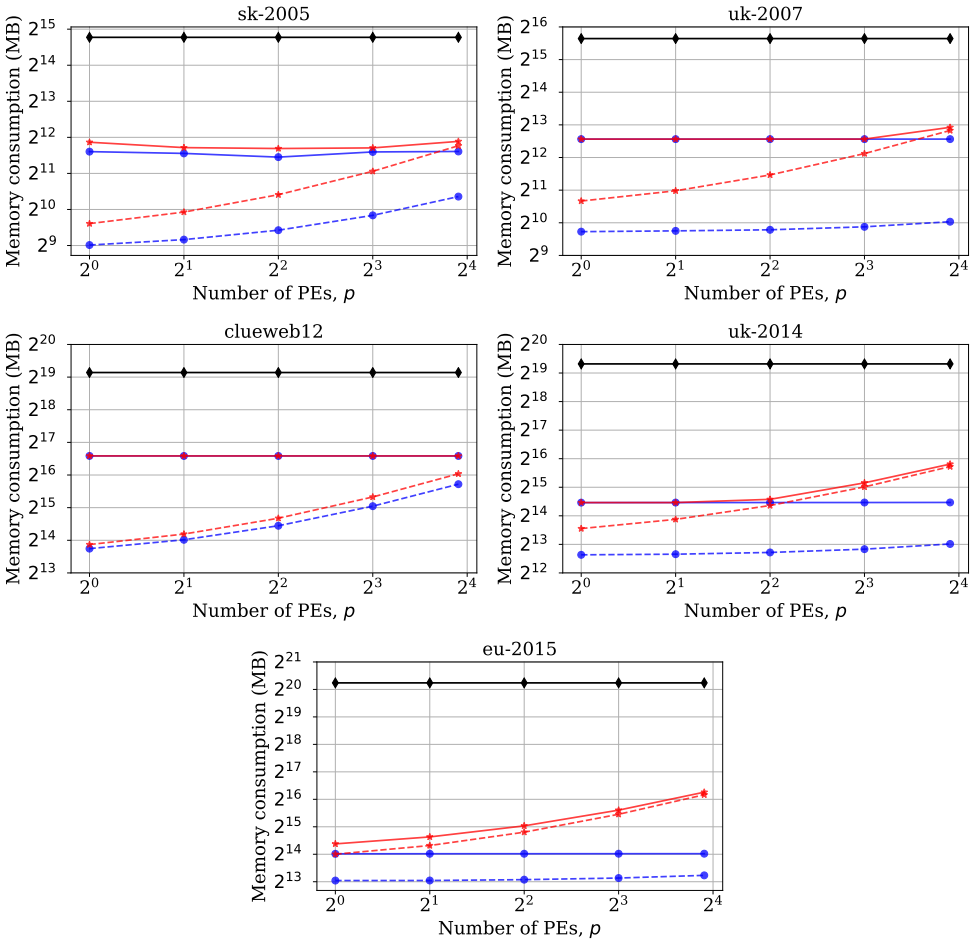


Figure 5.14: Memory consumption of parallel LPAs. Here the black curve is the memory consumption of the graph in the internal memory.

5.4 Conclusion and Future Work

We presented algorithms that are able to partition and cluster huge complex networks with billions of edges on cheap commodity machines. This has been achieved by using

a semi-external variant of the size-constrained LPA that can be used for coarsening and as a simple local search algorithm. A shared-memory parallelizations of the algorithm further reduces the running time. Moreover, we presented the first fully external graph clustering/partitioning algorithm that is able to deal with size constraints. Our experiments indicate that our semi-external algorithms are able to compute high-quality partitions in time comparable to that of efficient internal memory implementations. As a part of future work, it might be interesting to define a (semi-)external clustering algorithm that optimizes different metrics. This can be done by using the techniques presented in this chapter but using different update rules to compute the cluster IDs of vertices. For example, Hamann et al. [Ham+18] present a clustering algorithm that optimizes modularity using SSD disks. Moreover, it would be interesting to develop a size-constrained label propagation algorithm in the external memory using the count-min sketch by Graham and Muthukrishnan [CM05] to track clusters of size larger than $L_{\max}$.

Chapter 6

6 Fast Sparsification of Hypergraphs

We optimize a fast and high-quality multi-level algorithm that partitions hypergraphs into k balanced blocks. Hypergraphs are an extension of graphs that allow a single edge to connect *more than two* vertices. Thus, they describe models and processes more accurately additionally allowing more possibilities for improvement. Our algorithm sparsifies a hypergraph such that the resulting hypergraph can be partitioned significantly faster without loss in quality (or with insignificant loss).

To sparsify a hypergraph, we combine vertices that share a lot of hyperedges. In order to find such vertices in linear time with guaranteed probability, we use the min-hash technique by Broder [Bro97]. But the ratios of shared hyperedges by two vertices depends on the input hypergraph. Moreover, even in one hypergraph the ratios of shared hyperedges by two neighbors may be heterogeneous. Therefore, we developed an adaptive sparsifier that repeatedly applies the min-hash technique to find pairs of neighbors with ratios of shared neighbors within a preset interval.

Contribution: We developed a fast linear-time hypergraph sparsification algorithm within the hypergraph partitioning framework `KaHyPar` (Karlsruhe Hypergraph Partitioning) [Akh+17]. On average, `KaHyPar` with sparsifier performs partitioning about 1.5 times faster preserving quality if hyperedges are large.

The remaining of this chapter is organized in the following way. We discuss the related work and explain details of our sparsifier in Section 6.2. Section 6.3 presents experimental evaluation of our sparsifier.

Reference. This chapter is based on the conference paper [Akh+17] published together with Tobias Heuer, Peter Sanders, and Sebastian Schlag. The text was written by Yaroslav Akhremtsev and Sebastian Schlag with the editing by Peter Sanders. The design and analyses of the algorithms were made by Yaroslav Akhremtsev, Sebastian Schlag, and Peter Sanders. The algorithms were implemented by Yaroslav Akhremtsev.

6.1 Preliminaries

Let $H = (V, E, c, \omega)$ is undirected hypergraph where $V = \{0, \ldots, n-1\}$ is a set of vertices, $E = \{e : e \subset V\}$ is a set of hyperedges (or nets), $c : V \to \mathbb{R}_{>0}$ is a vertex weight function, and $\omega : E \to \mathbb{R}_{>0}$ is a net weight function. The vertices of a net are called *pins* [CA99b]. P denotes the multiset of all pins in H. We extend edge and vertex weight functions ω and c to sets, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. $\mathrm{I}(v) = \{e \in E : v \in e\}$ denotes the set of all incident nets of v. The *degree* of a vertex v is $d(v) = |\mathrm{I}(v)|$. We denote the neighbors of v as $N(v) = \{u \in V : |I(u) \cap I(v)| > 0\}$. A *k-way partition* of a hypergraph H is a partition of its vertex set into k disjoint *blocks* $V_1, \ldots, V_k$ such that $\bigcup_{i=1}^{k} V_i = V$. A k-way partition is *ε-balanced* if each block V_i satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1+\varepsilon)\lceil c(V)/k \rceil$ for some ε. A block V_i is *overloaded* if $c(V_i) > L_{\max}$ and *underloaded* if $c(V_i) < L_{\max}$. The number of pins of a net e in block V_i is $\Phi(e, V_i) := |\{v \in V_i \mid v \in e\}|$. The *connectivity set* of e is $\Lambda(e) := \{V_i \mid \Phi(e, V_i) > 0\}$. The *connectivity* of a net e is $\lambda(e) := |\Lambda(e)|$. A cut net is a net that has connectivity greater than one. We denote the set of cut nets as E_c.

The solution of the *k-way hypergraph partitioning problem* is an ε-balanced k-way partition of a hypergraph H that minimizes an objective function over the cut nets. The two most commonly used objective functions are the *cut-net* metric: $\text{cut} := \sum_{e \in E_c} \omega(e)$ and the *connectivity* metric $(\lambda{-}1) := \sum_{e \in E_c} (\lambda(e){-}1)\, \omega(e)$. In this chapter, we use the connectivity-metric, which accurately models the total communication volume of parallel sparse matrix-vector multiplication [CA99b]. The k-way hypergraph partitioning problem that optimizes any of these two objective functions is NP-hard [Len90].

Contraction is a construction of a new hypergraph with respect to a clustering of a hypergraph. The vertices of the new hypergraph represent clusters and edges are induced by connectivity between clusters. The vertex weights are set to the weight of the corresponding cluster and edge weights are equal to the weight of the edges that run between the respective clusters. More formally, given a graph clustering $V_1, \ldots, V_k$, the contracted hypergraph is defined as $H' = (V', E', c', \omega')$, where $V' = \{1, \ldots, k\}$ and E' consist of nets $e \in E$ such that $\lambda(e) > 1$ and all pins in e are replaced by the number of cluster $(0, \ldots, k-1)$ they belong to. Furthermore, $c'(i) = \sum_{v \in V_i} c(v),\ i \in V'$ and parallel nets are merged into one net whose weight is set to the sum of weights of merged nets.

6.2 Min-Hash Based Pin Sparsifier

The multi-level hypergraph partitioning framework `KaHyPar` employs algorithms that perform some computations on vertices and their set of neighbors. This requires an iteration over the set of all pins for all incident nets. For hypergraphs with many

large nets, these calculations can therefore have a significant impact on the overall running time. To alleviate this impact, we developed a pin sparsifier. The central idea is to identify and contract vertices that share many nets ("close" vertices), while leaving "distant" vertices untouched. The *distance* between two vertices v and w is hereby defined as $D(v, w) = 1 - J(\mathrm{I}(v), \mathrm{I}(w))$, where $J(X, Y) = |X \cap Y|/|X \cup Y|$ is the Jaccard index of sets X, Y. Calculating these distances for each vertex v and all of its neighbors $N(v)$ would lead to a quadratic algorithm. We therefore use the *min-hash*-based fingerprints [Bro97]. Deveci et al. [DKc13] adapt this idea to identify similar hyperedges (i.e., to build a net sparsifier), while Haveliwala et al. [HGI00] use it to cluster URLs. Both papers do not give a generic algorithm without having to choose several parameters manually. We present our adaptive approach to determine these parameters dynamically. We also extend Haveliwala et al.'s approach such that the resulting clusters are more balanced.

Suppose we have a set Σ of all possible permutations of elements of a set U. Then we can define the *min-hash* family $\mathcal{H} = \{h_\sigma(\mathrm{I}(v)) = \min\{\sigma(e)|e \in \mathrm{I}(v)\}|\sigma \in \Sigma\}$. The *min-hash* family of hash functions is known to be locality sensitive [GIM99; IM98]. To prove this, we first give the definition of (R, cR, P_1, P_2)-sensitive families of hash functions.

Definition 6.1

A family of hash functions $\mathcal{H} : U \mapsto U$ is called (R, cR, P_1, P_2)-sensitive if $\forall\ p, q \in S$ the following conditions are true:

(i) if $D(p, q) \leq R$ then $\forall\ h \in \mathcal{H} :\ Pr[h(p) = h(q)] \geq P_1$

(ii) if $D(p, q) \geq cR$ then $\forall\ h \in \mathcal{H} :\ Pr[h(p) = h(q)] \leq P_2$

Here $D(p, q)$ is a distance function between objects p and q.

Now we prove that the *min-hash* family is (R, cR, P_1, P_2)-sensitive. Although it is a well-known fact, we were not able to find a detailed proof of it. Therefore, we present it in Theorem 6.2. Note, that we are interested in families of hash functions such that $P_1 - P_2 > 0$. This positive difference allows to distinguish between close and distant objects by applying hash functions to them.

Theorem 6.2

The min-hash family $\mathcal{H} = \{h_\sigma(X) = \min\{\sigma(e)|e \in X\}\}$ is (R, cR, P_1, P_2)-sensitive, where X and Y are finite sets with elements from a finite universe U and σ is a random permutation of all elements of U.

Proof. We prove that $Pr[h_\sigma(X) = h_\sigma(Y)] = J(X, Y)$.

$$Pr[h_\sigma(X) = h_\sigma(Y)] = \sum_{e \in X \cap Y} Pr[\sigma(e) = h_\sigma(X \cup Y)] = \frac{|X \cap Y|}{|X \cup Y|},$$

since for $e \in X \cup Y$

$$Pr[\sigma(e) = h_\sigma(X \cup Y)] = \frac{\binom{n}{k} \cdot (k-1)! \cdot (n-k)!}{n!} = \frac{1}{k},$$

where $n = |U|$ and $k = |X \cup Y|$.

Hence, if $D(X,Y) \leq R$ then $Pr[h_\sigma(X) = h_\sigma(Y)] = J(X,Y) \geq 1 - R = P_1$. Analogously, if $D(X,Y) \geq cR$ then $Pr[h_\sigma(X) = h_\sigma(Y)] \leq 1 - cR = P_2$. □

We define a set of fingerprints $\{g_i(x,k) = (h_{i,1}(x), h_{i,2}(x), \cdots, h_{i,k}(x))\}_{i=1\ldots l}$ where each hash function $h_{i,j}$ is chosen uniformly at random from the min-hash family $\mathcal{H}$. Note that to represent a random permutation one needs $\Theta(|U| \log |U|)$ bits. Therefore, in order to make our algorithm practical, we can either use hash functions or pseudo-random permutations [Knu98, Section 3.4.2, Page 12] instead of random permutations. We use hash functions like Haveliwala et al. [HGI00] and Broder et al. [Bro+97] since this approach has better cache locality. Specifically, we use MurmurHash [Mur] hash functions that do not have theoretical guarantees (that is, a MummurHash hash function does not yield a random permutation) but still provide good results in practice.

During sparsification, we want vertices with the same fingerprint to be assigned to the same cluster. Two fingerprints are equal, if and only if all k hash values are equal. Theorem 6.3 proves the bounds on the probability of two vertices to have equal fingerprints using the fact that *min-hash* family is (R, cR, P_1, P_2)-sensitive. Since these fingerprints approximate the distance between two vertices, the size of the fingerprint (i.e., the number k of hash values) affects the probability that two vertices are assigned to the same cluster. By increasing the number of hash values, we decrease the probability that two "distant" vertices have the same fingerprint. More precisely, if $D(v,w) \geq cR$ then $Pr[g(x) = g(y)] \leq (1 - cR)^k$. As we can see, the probability of two vertices to have equal fingerprints *exponentially* decreases as the size of a fingerprint increases. However, at the same time, this also decreases the probability of "close" vertices to be in the same cluster. To avoid this problem, we calculate $l > 1$ fingerprints for each vertex.

Theorem 6.3

Consider a fingerprint $g(x) = (h_1(x), h_2(x), \cdots, h_k(x))$, where all k hash functions are chosen uniformly at random from a min-hash family $\mathcal{H}$. Then the following probabilities hold:

(i) if $D(x,y) \leq R$ then $Pr[g(x) = g(y)] \geq P_1^k$

(ii) if $D(x,y) \geq cR$ then $Pr[g(x) = g(y)] \leq P_2^k$

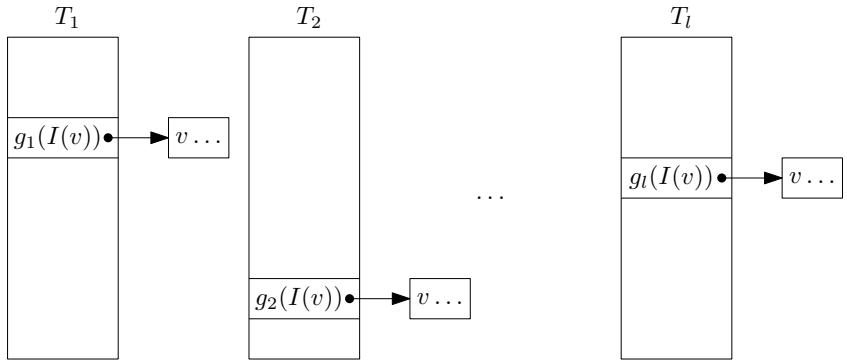


Figure 6.1: An outline of hash tables $T_1, \ldots, T_l$.

Proof. We prove the first case. The second case is proven analogously. Since all k hash functions $h_1(x), \cdots, h_k(x)$ are independent then

$$Pr[g(x) = g(y)] = Pr[\bigwedge_{j=1\ldots k} h_j(x) = h_j(y)] = \prod_{j=1\ldots k} Pr[h_j(x) = h_j(y)] \geq P_1^k$$

□

Sparsification

To sparsify a hypergraph $H = (V, E)$, we want to cluster its vertices such that "close" vertices belong to the same cluster. In order to do this, we build a set of l hash tables $T_1, \ldots, T_l$ by inserting a vertex $v \in V$ in the buckets that corresponds to a key $g_i(\mathrm{I}(v))$ in $T_1, \ldots, T_l$. Figure 6.1 outlines the resulting hash tables $T_1, \ldots, T_l$. One hash table can be constructed in $\mathcal{O}(k \cdot \sum_{v \in V} d(v)) = \mathcal{O}(k|P|)$ time, where $d(v)$ corresponds to the time to calculate one *min-hash* function. Let $T_i[v]$ denote a bucket such that $\forall u \in T_i[v] : g_i(\mathrm{I}(v)) = g_i(\mathrm{I}(u))$. By representing $T_i[v]$ as a hash table, we can perform insert and delete operations in $\mathcal{O}(1)$ expected time. Further in this section all time bounds are for the expected time. The clustering algorithm runs a BFS traversal on an implicit undirected graph $\mathcal{G} = (V, \mathcal{E})$. $\mathcal{G}$ is represented by $T_1, \ldots, T_l$ in the following way: an edge $(v, u) \in \mathcal{E}$ if $\exists\ i \in [1, l] : g_i(\mathrm{I}(v)) = g_i(\mathrm{I}(u))$. Figure 6.2 shows an example of such a graph. The BFS traversal visits the vertices of the same connected component and adds them to the same cluster as long as the maximum cluster size $c_{\max}$ is not exceeded. When the algorithm visits a vertex it removes it from all l hash tables. The corresponding pseudocode is shown in Algorithm 6.1. The time for visiting a vertex is the time to access l hash tables that is $\mathcal{O}(kl)$. Further, we need to remove the vertex from l hash tables. This can be done in $\mathcal{O}(kl)$ time as well. Due to removal of visited vertices, the running time of the BFS traversal is $\mathcal{O}(kl|V|)$.

Quality and speed of our algorithm depend on the choice of parameters k and l. Shakhnarovich et al. [SDI06] choose k and l using a fixed global distance. We describe

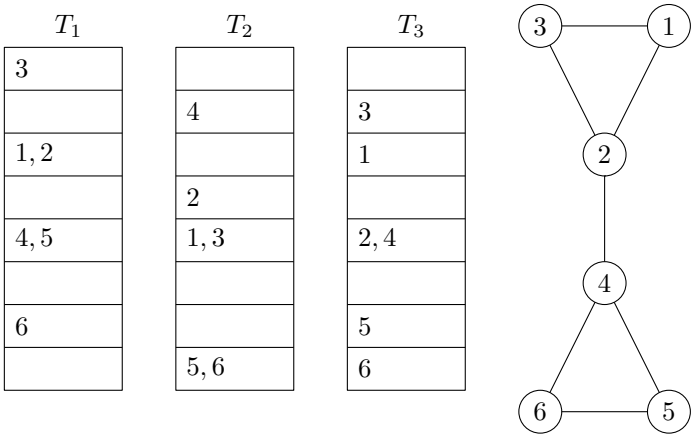


Figure 6.2: An example of hash tables $T_1, \ldots, T_l$ and corresponding implicit graph $\mathcal{G}$.

Algorithmus 6.1: The implicit BFS traversal.

Input: Graph $G = (V, E)$, A set of hash tables $\{T_1, \ldots, T_l\}$
Output: Array cluster // *Array of size $|V|$ to store cluster ids of vertices*

```
1  Function ImpicitBFS
2      Array visited                    // Array of size |V| to mark visited vertices
3      foreach v ∈ V do
4          if visited[v] then continue
5          Queue q                                                    // FIFO
6          visited[v] = true
7          q.push(v)
8          while q is not empty do
9              w ← q.pop()
10             for i ← 1 to l
11                 foreach u ∈ T_i[w] do
12                     if not visited[u] and max cluster size is not exceeded then
13                         visited[u] = true
14                         cluster[u] = cluster[v]
15                         q.push(u)
16                         remove u from T_1[w] ... T_l[w]
17                         if max cluster size is exceeded then go to 3
18             remove w from T_1[w] ... T_l[w]
```

the details of their approach in the following paragraph. Furthermore, since the distance between vertices varies in different parts of a hypergraph, we describe how to adaptively choose both k and l for each vertex individually in Section 6.2.1. Haveliwala et al. [HGI00] assign vertices of one bucket to one cluster without a possibility to extend this cluster; whilst our algorithm computes a more balanced clustering by extending a cluster such that its size is at least $c_{\min}$ (min cluster size) and at most max cluster size ($c_{\max}$). Furthermore, Haveliwala et al. do not give any algorithmic solution to select the parameters k and l. Therefore, our algorithm produces more balanced clusterings, which have less unit size clusters and less "large" clusters. A big number of unit size clusters results in a coarsened graph – built using the clustering – that is not small enough to speed up the hypergraph partitioning algorithm. But the presence of "large" clusters prevents to find a "good" partition where vertices from a "large" cluster belong to different blocks.

Choice of parameters. Shakhnarovich et al. find the parameter l such that for any pair of objects $p, q \in S : \; D(p,q) \leq R \; P[\exists \; i \in [1,l] : g_i(p) = g_i(q)] \geq 1 - \delta_1$. Furthermore, $P[\exists \; i \in [1,l] : g_i(p) = g_i(q)] \geq 1 - (1 - P_1^k)^l \geq 1 - \delta_1$ and, thus, $l \geq \log_{1-P_1^k} \delta_1$. To find the parameter k, Shakhnarovich et al. build a data structure from a sample $S' \subset S$ and choose k which minimizes the running times of the sample of queries. We employ the same inequality to choose l, but choose k such that $\forall \; p, q \in S$ and $D(p,q) \geq cR : \forall \; i \in [1,l] \; P[g_i(p) = g_i(q)] \leq \delta_2$. Since $P[g_i(p) = g_i(q)] \leq P_2^k \leq \delta_2$. Thus, $k \geq \log_{P_2} \delta_2$. Unfortunately, to find the parameters k and l, we need to know the probabilities P_1 and P_2, which depend on R. We tried to estimate the average value of R by sampling vertices and calculating distances from them to their neighbors. Unfortunately, the resulting partitioning of the coarsened graph – built using the clustering of G – usually does not yield a good partitioning.

6.2.1 Adaptive Clustering

The adaptive clustering works as follows: In the i-th iteration we build the hash table T_i by inserting each vertex v using a fingerprint $g_i(\mathrm{I}(v), k(v))$ as a key, where $k(v)$ is the size of the fingerprint of v. Next, we use T_i to extend clusters that were previously built using the hash tables $T_1, \ldots, T_{i-1}$. We now present our algorithm in more detail.

First, we describe an adaptive construction of a hash table using a variable number of hash functions. Bawa et al. [BCG05] use a similar idea to answer nearest-neighbors queries. Next we describe an adaptive clustering algorithm that incrementally builds a clustering by constructing a new hash table on each iteration and using it to extend clusters.

Adaptive construction of hash table H. Bawa et al. [BCG05] use a similar idea to answer nearest-neighbors queries. We build H incrementally using a fingerprint $g(x, k)$ for an input set of vertices $\mathcal{V}$ (which is not necessary all vertices). During each iteration for each vertex $v \in \mathcal{V}$ we increment $k(v)$ and reinsert it with a key $g(\mathrm{I}(v), k(v))$; if $|H[v]| \leq c_{\max}$ and $k(v) \geq h_{\min}$ then we finished to compute the fingerprints of $H[v]$ and we remove them from $\mathcal{V}$. This means that whole bucket $|H[v]|$ can be assigned to one cluster and there is no need to reduce its size by increasing the size of the fingerprint of v. We repeat this process until fingerprints of all vertices are computed. Algorithm 6.2 shows the pseudocode that constructs a hash table H. Our algorithm builds H in $\mathcal{O}(h_{\max} \cdot \sum_{v \in \mathcal{V}} d(v)) = \mathcal{O}(h_{\max} \cdot |P|)$ time, where $h_{\max}$ is the maximum number of hash functions.

Algorithmus 6.2: The adaptive construction of a hash table.

Input: Set of vertices $\mathcal{V}$
Output: Hash table H

```
1  Function
2      k = 1
3      HashTable H
4      HashTable prevH
5      while V is not empty and k ≤ h_max do
6          HashTable curH
7          foreach v ∈ V do
8              curH.insert(g(I(v), k), v)
9          foreach v ∈ V do
               // prevH[v] and curH[v] are equal.
10             if |prevH[v]| = |curH[v]| and k ≥ h_min then
11                 V = V \ curH[v]
12                 foreach u ∈ curH[v] do
13                     H.insert(g(I(u), k), u)
               // Cluster of v is small enough
14             if v ∈ V and |curH[v]| ≤ c_max and k ≥ h_min then
15                 V = V \ curH[v]
16                 foreach u ∈ curH[v] do
17                     H.insert(g(I(u), k), u)
18         k = k + 1
19         exchange curH and prevH
       // Copy the rest of V
20     foreach v ∈ V do
21         H.insert(g(I(v), k), v)
```

Adaptive clustering algorithm (ACA). Suppose that we have already performed ACA for $i-1$ hash tables and we want to build and process the hash table T_i. We consider the set of active vertices that contains vertices which belong to clusters with sizes less than $c_{\min}$. Note that when we construct the hash table T_1 all vertices are active. First, we build the hash table T_i from the set of active vertices. Next, we process the hash table T_i as follows. For each active vertex v, we iterate over the active vertices in $T_i[v]$, assign them to cluster c_v of v while the size of c_v is less than $c_{\max}$ and remove them from $T_i[v]$. If the size of c_v exceeds $c_{\min}$ then all vertices in c_v become inactive. We introduce parameters $c_{\min}$ and $c_{\max}$ to build a more balanced clustering. The algorithm stops if the number of clusters is less than $|V|/2$ or it exceeds the maximum number of hash tables l. We process a hash table in $\mathcal{O}(h_{\max} \cdot |V|)$ time, since we look up a bucket of a vertex in $\mathcal{O}(h_{\max})$ time and visit each vertex constant number of times. In summary, ACA works in $\mathcal{O}(l \cdot h_{\max} \cdot |P|)$ time since we have at most l hash tables. Note that this algorithm still needs an input parameter $h_{\min}$. This parameter means that any two vertices v, w cannot be in the same bucket if $g(\mathrm{I}(v), k) = g(\mathrm{I}(w), k)$ and $k < h_{\min}$. Thus, in order for v and w to be in the same bucket they must be sufficiently "close". Namely, let $D(v, w) = R$ then $P[g(\mathrm{I}(v), h_{\min}) = g(\mathrm{I}(w), h_{\min})] = (1-R)^{h_{\min}} \geq \delta$ and $R \leq 1 - \delta^{1/h_{\min}}$. For example, if $h_{\min} = 20$ then for two vertices to be in the same bucket with probability at least 0.5 they must be at distance at most 0.04. The problem is that if $h_{\min}$ is small and a hypergraph has a small average distance than even vertices with distance greater than the average distance may be in the same bucket; this results in a "bad" clustering. For example, if $h_{\min} = 2$ then the probability of two vertices at distance 0.5 to be in the same bucket is 0.25. If $h_{\min}$ is large and a hypergraph has a large average distance than a lot of clusters will have a unit size. For example, if $h_{\min} = 20$ then the probability of two vertices at distance 0.1 to be in the same bucket is $(1-0.1)^{20} < 0.13$. In our experiments we set c_{min} to 2, c_{max} to 10, h_{min} to 10, h_{max} to 100 and l to 5.

6.3 Experiments

The algorithm is implemented in the n-level hypergraph partitioning framework `KaHyPar` [Kah] (**Ka**rlsruhe **Hy**pergraph **Par**titioning). The code is written in C++ and compiled using g++-5.2 with flags `-O3 -mtune=native -march=native`.

System. All experiments are performed on a single core of a machine consisting of two Intel Xeon E5-2670 Octa-Core processors (Sandy Bridge) clocked at 2.6 GHz. The machine has 64 GB main memory, 20 MB L3-Cache and 8x256 KB L2-Cache and is running Ret Hat Enterprise Linux (RHEL) 7.2.

Instances. We evaluate our algorithm on a large collection of hypergraphs, which contains instances from three benchmark sets: the ISPD98 VLSI Circuit Benchmark Suite [Alp98], the University of Florida Sparse Matrix Collection [DH11], and the

international SAT Competition 2014 [Bel+14]. Sparse Matrices are translated into hypergraphs using the row-net model [CA99b]; that is, each row corresponds to a net and each column corresponds to a vertex. For SAT instances, each Boolean variable (and its complement) is mapped to a vertex and each clause corresponds to a net [PM07]. To evaluate the pin sparsifier, we use 294 hypergraphs [GM16]. In each case, the hypergraphs are partitioned into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks with $\varepsilon = 0.03$. For each value of k, a k-way partition is considered to be *one* test instance, resulting in a total of 1342 instances for the benchmark set (note that for 16 instances we stopped calculations because the time limit was exceeded).

Methodology. We perform ten repetitions with different seeds for each test instance and report the *arithmetic mean* of the computed $(\lambda - 1)$-cut and running time. We use the *geometric mean* to calculate average running time and cut size for different instances. Furthermore, we use *harmonic mean* to calculate average speed-up of `KaHyPar` with the sparsifier over `KaHyPar` without. Note that we exclude hypergraphs with zero cut sizes since we cannot include them in the calculation of the geometrical mean.

Evaluation of Sparsifier. Our sparsifier is intended to improve the running time for hypergraphs with large nets. As can be seen in Figure 6.3 and in Table 6.1, the sparsifier speeds up the partitioning process if the median net size $|\tilde{e}|$ is large ($|\tilde{e}| \geq 28$). Specifically, 83.6% of instances have speed-up greater than 1. Furthermore, the harmonic mean speed up of the partitioner is 1.3. But the sparsifier slows down a bit the partitioning process if the median net size $|\tilde{e}|$ is small. Nevertheless, the Wilcoxon signed-rank test (see Section 2.3.3) show that the differences between the running times of `KaHyPar` with and without the sparsifier are statistically significant. Moreover, the Wilcoxon signed-rank test show that the difference between cut sizes of `KaHyPar` with and without the sparsifier is statistically significant when $|\tilde{e}| \geq 28$ and statistically *in*significant when $|\tilde{e}| < 28$. Thus, the sparsification of hypergraphs improves quality when $|\tilde{e}| \geq 28$.

Note that we select the threshold 28 considering the experiments on the full benchmark set of instances. However, we would have observed a similar behavior on any random subset of these instances. Therefore, we expect that results of our sparsifier with this threshold will be similar on other benchmark sets.

To show the correlation between the median net size and speed up even further, we present Figure 6.4 and Figure 6.5. The maximum harmonic speed-up of 1.7 is achieved on the graphs with $|\tilde{e}|$ greater that 108. Furthermore, 90.4% of the hypergraphs with $|\tilde{e}|$ greater than 36 have speed-ups greater than 1.

Table 6.1: Effects of applying the pin sparsifier using all instances.

		use sparsifier		
		never	always	if $\|\tilde{e}\| \geq 28$
$\|\tilde{e}\| \geq 28$	cut	13156.7	**13155.2**	**13155.2**
	t(s)	20.0	**13.3**	**13.3**
	speed-up	1.0	**1.3**	**1.3**
$\|\tilde{e}\| < 28$	cut	**7764.4**	7770.5	**7764.4**
	t(s)	**14.8**	15.0	**14.8**
	speed-up	**1.0**	0.9	**1.0**

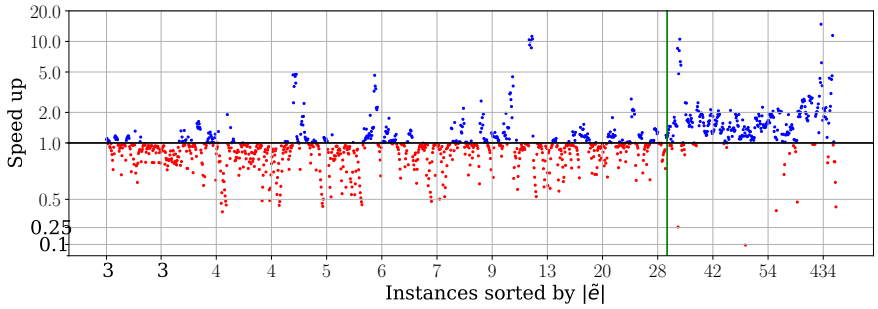


Figure 6.3: Threshold for sparsifier.

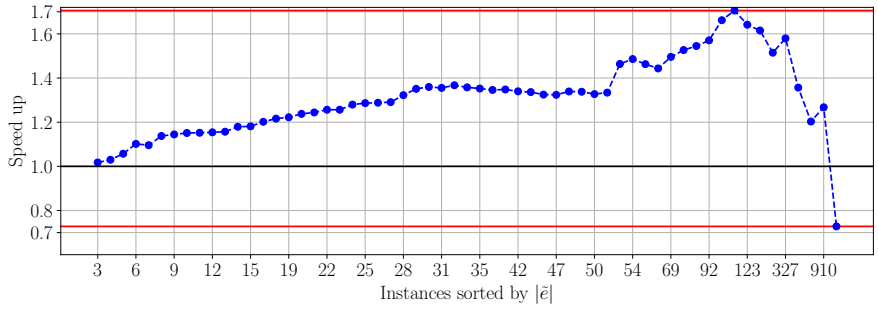


Figure 6.4: *Cumulative* harmonic mean speed-ups for different minimum median net sizes. Here a point (x, y) means that the harmonic mean speed-up of the graphs with $|\tilde{e}| \geq x$ is y.

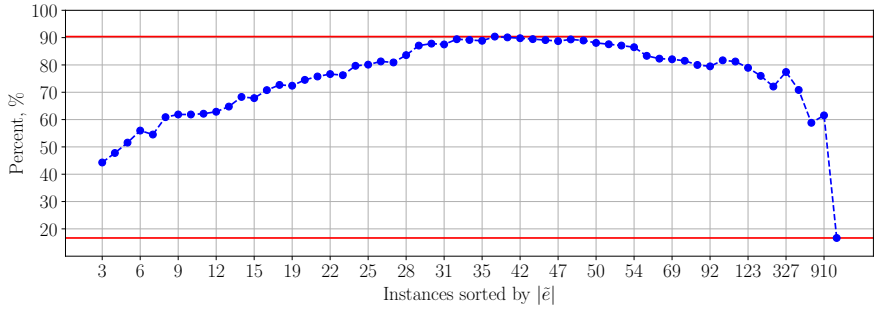


Figure 6.5: *Cumulative* percentage of graphs with speed-ups greater than 1 for different minimum median net sizes. Here a point (x, y) means that $y\%$ of the graphs with $|\tilde{e}| \geq x$ have speed-ups greater than 1

6.4 Conclusion and Future Work

We developed a linear time hypergraph sparsifier based on the min-hash technique that speeds up computation of multi-level hypergraph partitioning framework `KaHyPar` by a factor of 1.5 on average if hyperedges are large. Moreover, our sparsifier does not affect quality of resulting partitions which is confirmed by the Wilcoxon signed-rank test.

As a part of future work, we are planning to parallelize the sparsifier since it processes all vertices independently. Specifically, we can parallelize the construction of hash tables by computing fingerprints of vertices in parallel. Furthermore, we can parallelize the adaptive clustering by assigning vertices to clusters in parallel.

Another interesting idea for future work is an adaptive clustering algorithm that adds vertices to a cluster according to their connectivity to it.

It would be interesting to consider other ways to compute fingerprints of vertices and how they affect resulting quality and running times. Specifically, one can consider choosing an edge ID as a hash value with probability proportional to the size of the edge. Thus, if two vertices adjacent to a large edge it is likely that they will end up in the same cluster. Another approach it to cluster vertices according to k smallest hash values instead of one minimum hash value. Specifically, if k smallest hash values of two vertices are equal then they are assigned to the same cluster. This approach allows to use computed hash values in a more efficient way.

Another interesting application of our sparsifier is graph sparsification. Specifically, we can cluster vertices that share multiple neighbors and replace the clusters by single vertices. This approach can help to speed up processing of large social networks that have high degree vertices. First of all, such network can be contracted faster by reducing degrees of high degree vertices. Moreover, if the coarsest graph contains multiple high degree vertices this can significantly slow down initial partitioning phase. For example, this can improve speed-ups of our parallel graph partitioning framework `Mt-KaHIP`.

Chapter 7

7 Conclusion

Graph partitioning can be an important component of efficient large scale parallel graph algorithms. In this thesis, we mainly focus on shared-memory and external memory models that consider algorithms which are able to process large networks on a single machine. For the shared-memory model, we presented efficient multi-level graph partitioning algorithm that constructs high-quality partitions, guarantees balance, and shows good performance. The main components of our algorithm are parallel label propagation for coarsening and refinement and effective approach to parallelize localized local search. For the external memory model, we presented fast (semi-)external multi-level graph partitioning algorithm that constructs high-quality partitions and able to partition a graph with $80.5G$ edges. The main component of our algorithm is an efficient (semi-)external label propagation algorithm that allows efficiently to contract large graphs until they small enough to be partitioned by an internal memory partitioner.

Additionally, we considered a fast sparsification technique that can be used to speed-up hypergraph partitioning algorithm on hypergraphs with large hyperedges. This randomized technique finds vertices that have similar incident hyperedges and combines them in linear time such that the resulting graph can be partitioned faster without quality loss.

List of Algorithms

List of Figures

List of Tables

List of Theorems

Bibliography

[AB11] Bas Fagginger Auer and Rob H. Bisseling. "A GPU Algorithm for Greedy Graph Matching". In: *Facing the Multicore - Challenge II - Aspects of New Paradigms and Technologies in Parallel Computing [Proceedings of a conference held at the Karlsruhe Institute of Technology (KIT), September 28-30, 2011].* Pages 108–119. 2011. [see page 27]

[AB12] Bas Fagginger Auer and Rob H. Bisseling. "Graph coarsening and clustering on the GPU". In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings.* Edited by David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Volume 588. Contemporary Mathematics, page 223. ISBN: 978-0-8218-9038-7. American Mathematical Society, 2012. [see page 32]

[ABW02] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. "A Functional Approach to External Graph Algorithms". In: *Algorithmica* 32.3 (2002), pages 437–458. [see page 10]

[AK06] Amine Abou-Rjeili and George Karypis. "Multilevel algorithms for partitioning power-law graphs". In: *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece.* 2006. [see pages 25, 66]

[Akh11] Yaroslav Akhremtsev. "Design and Development of Functional Programming Language". Bachelor Thesis. Institute of Automatics and Computer Engineering, Moscow Power Engineering Institute, Russia. May 2011. [see page 223]

[Akh13] Yaroslav Akhremtsev. "Development and Analyzes of Shortest Paths Algorithms for Road Networks". Master Thesis. Institute of Automatics and Computer Engineering, Moscow Power Engineering Institute, Russia. May 2013. [see page 223]

[Akh+17] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. "Engineering a direct k-way Hypergraph Partitioning Algorithm". In: *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017.* Pages 28–42. 2017. [see pages 5, 175, 223]

[AL97] Cleve Ashcraft and Joseph WH Liu. “Using domain decomposition to find graph bisectors”. In: *BIT Numerical Mathematics* 37.3 (1997), pages 506–534. Springer. [see page 45]

[Alp98] C. J. Alpert. “The ISPD98 Circuit Benchmark Suite”. In: *Proceedings of the 1998 International Symposium on Physical Design*, pages 80–85. ISBN: 1-58113-021-X. Monterey, California, USA: ACM, 1998. [see page 183]

[AR04] Konstantin Andreev and Harald Räcke. “Balanced graph partitioning”. In: *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, pages 120–124. 2004. [see page 63]

[Arg03a] Lars Arge. “The Buffer Tree: A Technique for Designing Batched External Data Structures”. In: *Algorithmica* 37.1 (2003), pages 1–24. [see page 140]

[Arg03b] Lars Arge. “The Buffer Tree: A Technique for Designing Batched External Data Structures.” In: *Algorithmica* 37.1 (Oct. 31, 2003), pages 1–24. [see page 145]

[AS16] Yaroslav Akhremtsev and Peter Sanders. “Fast Parallel Operations on Search Trees”. In: *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*, pages 291–300. 2016. [see page 223]

[ASS15] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. “(Semi-)External Algorithms for Graph Partitioning and Clustering”. In: *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pages 33–43. 2015. [see pages 139, 223]

[ASS18] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. “High-Quality Shared-Memory Graph Partitioning”. In: *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 659–671. 2018. [see pages 45, 65, 223]

[Avi83] David Avis. “A survey of heuristics for the weighted matching problem”. In: *Networks* 13.4 (1983), pages 475–493. [see page 22]

[Axt+17] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. “In-Place Parallel Super Scalar Samplesort (IPSSSSo)”. In: *Proc. of the 25th ESA*, 9:1–9:14. 2017. [see page 79]

[Bad13] Michael Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*. Volume 9. Texts in Computational Science and Engineering. Springer, 2013. ISBN: 978-3-642-31045-4. DOI: 10.1007/978-3-642-31046-1. URL: https://doi.org/10.1007/978-3-642-31046-1. [see page 17]

[Bad+13a] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering*. 2013. [see pages 13, 14]

[Bad+13b] 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. Contemporary Mathematics. American Mathematical Society, 2013. ISBN: 978-0-8218-9038-7. DOI: 10.1090/conm/588. URL: https://doi.org/10.1090/conm/588. [see page 35]

[Bad+14] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. "Benchmarking for Graph Clustering and Partitioning". In: *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. 2014. [see pages 13, 14]

[BB87] Marsha J. Berger and Shahid H. Bokhari. "A Partitioning Strategy for Nonuniform Problems on Multiprocessors". In: *IEEE Trans. Computers* 36.5 (1987), pages 570–580. [see page 36]

[BCG05] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. "LSH Forest: Self-tuning Indexes for Similarity Search". In: *Proceedings of the 14th International Conference on World Wide Web*. WWW '05, pages 651–660. ISBN: 1-59593-046-9. Chiba, Japan: ACM, 2005. [see pages 181, 182]

[Bec+] Andreas Beckmann, Timo Bingmann, Roman Dementiev, Peter Sanders, and Johannes Singler. "STXXL Home Page". https://github.com/stxxl/stxxl. [see pages 140, 151]

[Bel+14] A. Belov, D. Diepold, M. Heule, and M. Järvisalo. *The SAT Competition 2014*. http://www.satcompetition.org/2014/. 2014. [see page 184]

[BH11] Una Benlic and Jin-Kao Hao. "A Multilevel Memetic Approach for Improving Graph k-Partitions". In: *IEEE Trans. Evolutionary Computation* 15.5 (2011), pages 624–642. [see page 17]

[Bir+13] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. "Efficient Parallel and External Matching". In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 659–670. 2013. [see pages 28, 66, 68, 80]

[BJ92] T. Nguyen Bui and C. Jones. "Finding good approximate vertex and edge partitions is NP-hard". In: *Information Processing Letters* 42.3 (1992), pages 153–159. Elsevier. [see page 63]

[Bou98] N Bouhmala. *Impact of different graph coarsening schemes on the quality of the partitions*. Technical report. Technical Report RT98/05-01, University of Neuchatel, Department of Computer Science, 1998. [see pages 18, 56]

[Bro97] A. Broder. “On the Resemblance and Containment of Documents”. In: *Proceedings of the Compression and Complexity of Sequences 1997*. SEQUENCES ’97, pages 21–. ISBN: 0-8186-8132-2. Washington, DC, USA: IEEE Computer Society, 1997. [see pages 5, 175, 177]

[Bro+97] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. “Syntactic clustering of the web”. In: *Computer Networks and ISDN Systems* 29.8-13 (1997), pages 1157–1166. Elsevier. [see page 178]

[BS11] C. Bichot and P. Siarry, editors. *Graph Partitioning*. Wiley, 2011. [see page 17]

[BS93] Stephen T. Barnard and Horst D. Simon. “A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems”. In: *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1993, Norfolk, Virginia, USA, March 22-24, 1993*, pages 711–718. 1993. [see pages 17, 57]

[Bul+13] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. “Recent Advances in Graph Partitioning”. In: *Lecture Notes in Computer Science* 9220 (2013), pages 117–158. [see pages 1, 17, 34]

[BV04] P. Boldi and S. Vigna. “The WebGraph Framework I: Compression Techniques”. In: *Proc. of the 13th Int. World Wide Web Conference*, pages 595–601. 2004. [see page 13]

[CA99a] U. V. Catalyurek and C. Aykanat. “Hypergraph-partitioning based Decomposition for Parallel Sparse-Matrix Vector Multiplication”. In: *IEEE Transactions on Parallel and Distributed Systems* 10.7 (1999), pages 673–693. IEEE. [see page 67]

[CA99b] Ü. V. Catalyürek and C. Aykanat. “Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication”. In: *IEEE Transactions on Parallel and Distributed Systems* 10.7 (1999), pages 673–693. ISSN: 1045-9219. [see pages 176, 184]

[CG97] Boris V. Cherkassky and Andrew V. Goldberg. “On Implementing the Push-Relabel Method for the Maximum Flow Problem”. In: *Algorithmica* 19.4 (1997), pages 390–410. [see page 47]

[Chi+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. “External-Memory Graph Algorithms”. In: *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California.* Pages 139–149. 1995. [see pages 140, 144]

[CL98] Jason Cong and Sung Kyu Lim. “Multiway partitioning with pairwise movement”. In: *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1998, San Jose, CA, USA, November 8-12, 1998*, pages 512–516. 1998. [see page 43]

[CM05] Graham Cormode and S. Muthukrishnan. “An improved data stream summary: the count-min sketch and its applications”. In: *J. Algorithms* 55.1 (2005), pages 58–75. [see page 174]

[CP08] C. Chevalier and F. Pellegrini. “PT-Scotch: A Tool for Efficient Parallel Graph Ordering”. In: *Parallel Computing* (2008), pages 318–331. [see pages 60, 67]

[CS11] Jie Chen and Ilya Safro. “Algebraic Distance on Graphs”. In: *SIAM J. Scientific Computing* 33.6 (2011), pages 3468–3490. [see page 24]

[Dav] T. Davis. *The University of Florida Sparse Matrix Collection.* [see pages 13, 14]

[DD96] Shantanu Dutt and Wenyong Deng. “VLSI circuit partitioning by cluster-removal using iterative improvement techniques”. In: *ICCAD*, pages 194–200. 1996. [see page 44]

[Del+12] Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato Fonseca F. Werneck. “Exact Combinatorial Branch-and-Bound for Graph Bisection”. In: *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*, pages 30–44. 2012. [see page 34]

[Dem06] Roman Dementiev. “Algorithm engineering for large data sets”. PhD thesis. Saarland University, 2006. URL: http://d-nb.info/983672970. [see page 11]

[DF11] Benjamin Doerr and Mahmoud Fouz. “Asymptotically Optimal Randomized Rumor Spreading”. In: *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, pages 502–513. 2011. [see page 62]

[DH03a] Doratha E. Drake and Stefan Hougardy. “A simple approximation algorithm for the weighted matching problem”. In: *Inf. Process. Lett.* 85.4 (2003), pages 211–213. [see page 22]

[DH03b] Doratha E. Drake and Stefan Hougardy. “Improved Linear Time Approximation Algorithms for Weighted Matchings”. In: *Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques, 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2003 and 7th International Workshop on Randomization and Approximation Techniques in Computer Science, RANDOM 2003, Princeton, NJ, USA, August 24-26, 2003, Proceedings*, pages 14–23. 2003. [see pages 22, 23]

[DH03c] Doratha E. Drake and Stefan Hougardy. “Linear Time Local Improvements for Weighted Matchings in Graphs”. In: *Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland, May 26-28, 2003, Proceedings*, pages 107–119. 2003. [see page 23]

[DH11] T. A. Davis and Y. Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Transactions on Mathematical Software* 38.1 (2011), 1:1–1:25. ISSN: 0098-3500. New York, NY, USA: ACM. [see page 183]

[DH72] W.E. Donath and A.J. Hoffman. “Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices”. In: *IBM Technical Disclosure Bulletin* 15.3 (1972), pages 938–944. [see page 17]

[DH73] William E Donath and Alan J Hoffman. “Lower bounds for the partitioning of graphs”. In: *IBM Journal of Research and Development* 17.5 (1973), pages 420–425. IBM. [see page 17]

[DH97] Anthony Christopher Davison and David Victor Hinkley. *Bootstrap methods and their application*. Volume 1. Cambridge university press, 1997. [see page 74]

[Die+00] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. “Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM”. In: *Parallel Computing* 26.12 (2000), pages 1555–1581. [see pages 36, 66]

[DKc13] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. “Hypergraph Sparsification and Its Application to Partitioning”. In: *Proceedings of the 2013 42Nd International Conference on Parallel Processing*. ICPP ’13, pages 200–209. ISBN: 978-0-7695-5117-3. Washington, DC, USA: IEEE Computer Society, 2013. [see page 177]

[DM98] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pages 46–55. IEEE. [see pages 32, 33]

[DMP94] Ralf Diekmann, Burkhard Monien, and Robert Preis. “Using helpful sets to improve graph bisections”. In: *Workshop on Interconnection Networks and Mapping and Scheduling Parallel Computations, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, February 7-9, 1994*, pages 57–74. 1994. [see page 46]

[DP10] Ran Duan and Seth Pettie. “Approximating Maximum Weight Matching in Near-Linear Time”. In: *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 673–682. 2010. [see page 24]

[DP14] Ran Duan and Seth Pettie. “Linear-Time Approximation for Maximum Weight Matching”. In: *J. ACM* 61.1 (2014), 1:1–1:23. [see page 24]

[Dur+14] Erika Duriakova, Neil Hurley, Deepak Ajwani, and Alessandra Sala. “Analysis of the semi-synchronous approach to large-scale parallel community finding”. In: *Proceedings of the second ACM conference on Online social networks, COSN 2014, Dublin, Ireland, October 1-2, 2014*, pages 51–62. 2014. [see pages 32, 33]

[Dut93] Shantanu Dutt. “New faster Kernighan-Lin-type graph-partitioning algorithms”. In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 370–377. 1993. [see page 39]

[Eve+97] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. “Fast Approximate Graph Partitioning Algorithms”. In: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*. Pages 639–648. 1997. [see page 63]

[FF56] Lester R Ford and Delbert R Fulkerson. “Maximal flow through a network”. In: *Canadian journal of Mathematics* 8.3 (1956), pages 399–404. [see page 47]

[Fie73] Miroslav Fiedler. “Algebraic connectivity of graphs”. In: *Czechoslovak mathematical journal* 23.2 (1973), pages 298–305. Institute of Mathematics, Academy of Sciences of the Czech Republic. [see page 17]

[Fie75] Miroslav Fiedler. “A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory”. In: *Czechoslovak Mathematical Journal* 25.4 (1975), pages 619–633. Institute of Mathematics, Academy of Sciences of the Czech Republic. [see page 17]

[Fjä98] Per-Olof Fjällström. *Algorithms for graph partitioning: A survey*. Volume 3. Linköping University Electronic Press Linköping, 1998. [see page 17]

[FK02] Uriel Feige and Robert Krauthgamer. “A Polylogarithmic Approximation of the Minimum Bisection”. In: *SIAM J. Comput.* 31.4 (2002), pages 1090–1118. [see page 63]

[FL93] Charbel Farhat and Michel Lesoinne. “Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics”. In: *International Journal for Numerical Methods in Engineering* 36.5 (1993), pages 745–764. Wiley Online Library. [see page 17]

[FM82] Charles M. Fiduccia and Robert M. Mattheyses. “A linear-time heuristic for improving network partitions”. In: *Proceedings of the 19th Design Automation Conference, DAC ’82, Las Vegas, Nevada, USA, June 14-16, 1982*, pages 175–181. 1982. [see pages 40, 71]

[Fun+18] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. “Communication-free Massively Distributed Graph Generation”. In: *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*. 2018. [see pages 14, 15]

[Gab90] Harold N. Gabow. “Data Structures for Weighted Matching and Nearest Common Ancestors with Linking”. In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California.* Pages 434–443. 1990. [see page 70]

[GBF11] Philippe Galinier, Zied Boujbel, and Michael Coutinho Fernandes. “An efficient memetic algorithm for the graph partitioning problem”. In: *Annals OR* 191.1 (2011), pages 1–22. [see page 17]

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. “Similarity Search in High Dimensions via Hashing”. In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB ’99, pages 518–529. ISBN: 1-55860-615-7. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. [see page 177]

[GJS74] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. “Some Simplified NP-Complete Problems”. In: *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 47–63. 1974. [see pages 1, 63]

[GL81] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981. ISBN: 0131652745. [see page 35]

[GM16] Michael T. Goodrich and Michael Mitzenmacher, editors. *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*. SIAM, 2016. ISBN: 978-1-61197-431-7. DOI: 10.1137/1.9781611974317. URL: https://doi.org/10.1137/1.9781611974317. [see page 184]

[GMR98] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. “The Queue-Read Queue-Write Asynchronous PRAM Model”. In: *Theor. Comput. Sci.* 196.1-2 (1998), pages 3–29. [see pages 9, 10]

[GMS14] Roland Glantz, Henning Meyerhenke, and Christian Schulz. “Tree-Based Coarsening and Partitioning of Complex Networks”. In: *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 364–375. 2014. [see page 24]

[GS94] Todd Goehring and Yousef Saad. *Heuristic algorithms for automatic graph partitioning*. Technical report. Citeseer, 1994. [see page 35]

[Gup97] Anshul Gupta. “Fast and effective algorithms for graph partitioning and sparse-matrix ordering”. In: *IBM Journal of Research and Development* 41.1.2 (1997), pages 171–183. IBM. [see pages 18, 22, 56]

[Hal+12] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. “Approximate weighted matching on emerging manycore and multithreaded architectures”. In: *IJHPCA* 26.4 (2012), pages 413–430. [see pages 30, 31]

[Ham+18] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. “Distributed Graph Clustering Using Modularity and Map Equation”. In: *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 688–702. 2018. [see page 174]

[HGI00] Taher Haveliwala, Aristides Gionis, and Piotr Indyk. “Scalable Techniques for Clustering the Web”. In: *In Proc. of the WebDB Workshop*, pages 129–134. 2000. [see pages 177, 178, 181]

[HH10] Sven Hanke and Stefan Hougardy. *New approximation algorithms for the weighted matching problem*. Forschungsinst. für Diskrete Mathematik, 2010. [see page 24]

[HL95a] Bruce Hendrickson and Robert W Leland. “A Multi-Level Algorithm For Partitioning Graphs.” In: *SC* 95.28 (1995), pages 1–14. [see pages 18, 42, 43, 56, 57]

[HL95b] Bruce Hendrickson and Robert W. Leland. “An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations”. In: *SIAM J. Scientific Computing* 16.2 (1995), pages 452–469. [see pages 17, 56]

[HM91] Juraj Hromkovic and Burkhard Monien. “The Bisection Problem for Graphs of Degree 4 (Configuring Transputer Systems)”. In: *Mathematical Foundations of Computer Science 1991, 16th International Symposium, MFCS'91, Kazimierz Dolny, Poland, September 9-13, 1991, Proceedings*, pages 211–220. 1991. [see page 46]

[Hoe04] Jaap-Henk Hoepman. “Simple Distributed Weighted Matchings”. In: *CoRR* cs.DC/0410047 (2004). [see page 27]

[Hoe63] Wassily Hoeffding. “Probability inequalities for sums of bounded random variables”. In: *Journal of the American statistical association* 58.301 (1963), pages 13–30. Taylor & Francis Group. [see pages 98, 201]

[HP10] JH Her and F Pellegrini. “Efficient and scalable parallel graph partitioning”. In: *Parallel Computing* (2010). [see pages 30, 55, 60, 67]

[HPD00] William W Hager, Soon Chul Park, and Timothy A Davis. “Block exchange in graph partitioning”. In: *Approximation and Complexity in Numerical Optimization*. Springer, 2000, pages 298–307. [see page 45]

[HPZ13] William W. Hager, Dzung T. Phan, and Hongchao Zhang. “An exact algorithm for graph partitioning”. In: *Math. Program.* 137.1-2 (2013), pages 531–556. [see page 35]

[HR73] Laurent Hyafil and Ronald L Rivest. *Graph partitioning and constructing optimal decision trees are polynomial complete problems*. IRIA. Laboratoire de Recherche en Informatique et Automatique, 1973. [see pages 1, 63]

[HSS10] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. “Engineering a scalable high quality graph partitioner”. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. 2010. [see pages 14, 24, 30, 54, 55, 61, 67]

[HSS18] Tobias Heuer, Peter Sanders, and Sebastian Schlag. "Network Flow-Based Refinement for Multilevel Hypergraph Partitioning". In: *17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L'Aquila, Italy*, 1:1–1:19. 2018. [see page 47]

[HW02] Jan Hungershöfer and Jens-Michael Wierum. "On the Quality of Partitions Based on Space-Filling Curves". In: *Computational Science - ICCS 2002, International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part III*, pages 36–45. 2002. [see page 17]

[IM98] Piotr Indyk and Rajeev Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality". In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC '98, pages 604–613. ISBN: 0-89791-962-9. Dallas, Texas, USA: ACM, 1998. [see page 177]

[IS86] Amos Israeli and Yossi Shiloach. "An Improved Parallel Algorithm for Maximal Matching". In: *Inf. Process. Lett.* 22.2 (1986), pages 57–60. [see page 27]

[JL96] Patrick Ciarlet Jr. and Françoise Lamour. "On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint". In: *Numerical Algorithms* 12.1 (1996), pages 193–214. [see page 35]

[JS98] Mark Jerrum and Gregory B. Sorkin. "The Metropolis Algorithm for Graph Bisection". In: *Discrete Applied Mathematics* 82.1-3 (1998), pages 155–175. [see pages 17, 56]

[Kab+17] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. "Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner". In: *PVLDB* 10.11 (2017), pages 1418–1429. [see page 223]

[Kah] URL: http://kahypar.org/. [see page 183]

[KBG12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. "GraphChi: Large-scale Graph Computation on Just a PC". In: *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. Volume 8, pages 31–46. 2012. [see page 139]

[Ker69] B. W. Kernighan. "Some graph partitioning problems related to program segmentation". PhD thesis. Princeton, 1969. [see page 71]

[Kim+11] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung Ro Moon. "Genetic approaches for graph partitioning: a survey". In: *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 473–480. 2011. [see pages 17, 56]

[KK95a] George Karypis and Vipin Kumar. "Analysis of Multilevel Graph Partitioning". In: *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, page 29. 1995. [see page 19]

[KK95b] George Karypis and Vipin Kumar. “Multilevel Graph Partitioning Schemes”. In: *Proceedings of the 1995 International Conference on Parallel Processing, Urbana-Champain, Illinois, USA, August 14-18, 1995. Volume III: Algorithms & Applications*. Pages 113–122. 1995. [see pages 18, 22, 43, 56, 57]

[KK96] George Karypis and Vipin Kumar. “Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, November 17-22, 1996, Pittsburgh, PA, USA*, page 35. 1996. [see pages 43–45, 60]

[KK97] George Karypis and Vipin Kumar. “A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm”. In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, Hyatt Regency Minneapolis on Nicollet Mall Hotel, Minneapolis, Minnesota, USA, March 14-17, 1997*. 1997. [see pages 30, 49, 54, 60]

[KK98a] G. Karypis and V. Kumar. “A fast and high quality multilevel scheme for partitioning irregular graphs”. In: *SIAM Journal on scientific Computing* (1998), pages 359–392. [see pages 18, 35, 56, 57]

[KK98b] George Karypis and Vipin Kumar. “A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering”. In: *J. Parallel Distrib. Comput.* 48.1 (1998), pages 71–95. [see page 60]

[KK98c] George Karypis and Vipin Kumar. “Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *J. Parallel Distrib. Comput.* 48.1 (1998), pages 96–129. [see pages 18, 56, 57]

[KK99] George Karypis and Vipin Kumar. “Parallel Multilevel series k-Way Partitioning Scheme for Irregular Graphs”. In: *SIAM Review* 41.2 (1999), pages 278–300. [see pages 29, 49, 60, 66, 90]

[KL70] Brian W Kernighan and Shen Lin. “An efficient heuristic procedure for partitioning graphs”. In: *The Bell system technical journal* 49.2 (1970), pages 291–307. Nokia Bell Labs. [see pages 37–39]

[Knu98] Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998. ISBN: 0201896842. URL: http://www.worldcat.org/oclc/312898417. [see page 178]

[KR13] S. Kirmani and P. Raghavan. “Scalable Parallel Graph Partitioning”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC’13)*, page 51. ISBN: 978-1-4503-2378-9. ACM, 2013. [see pages 63, 68]

[Kra+18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. 2018. [see page 136]

[KRC00] Stefan E. Karisch, Franz Rendl, and Jens Clausen. "Solving Graph Bisection Problems with Semidefinite Programming". In: *INFORMS Journal on Computing* 12.3 (2000), pages 177–191. [see page 35]

[Kri+10] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá. "Hyperbolic geometry of complex networks". In: *Physical Review E* (2010), page 036106. [see page 14]

[KSJ15] Andrei B. Khlopotine, Arun V. Sathanur, and Vikram Jandhyala. "Optimized Parallel Label Propagation Based Community Detection on the Intel(R) Xeon Phi(TM) Architecture". In: *27th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2015, Florianópolis, Brazil, October 17-21, 2015*, pages 9–16. 2015. [see page 33]

[KUW85] Richard M. Karp, Eli Upfal, and Avi Wigderson. "Constructing a Perfect Matching is in Random NC". In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 22–32. 1985. [see page 27]

[LaS+15] Dominique LaSalle, Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. "Improving graph partitioning for modern graphs and architectures". In: *Proceedings of the 5th Workshop on Irregular Applications - Architectures and Algorithms, IA3 2015, Austin, Texas, USA, November 15, 2015*, 14:1–14:4. 2015. [see pages 29, 52, 61]

[Lei14] F Thomson Leighton. *Introduction to parallel algorithms and architectures: Arrays · trees · hypercubes*. Elsevier, 2014. [see page 64]

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990. ISBN: 0-471-92838-0. [see page 176]

[Les] J. Leskovec. *Stanford Network Analysis Package (SNAP)*. [see pages 13, 14]

[Lib] URL: http://man7.org/linux/man-pages/man3/numa.3.html. [see page 88]

[LK13] D. LaSalle and G. Karypis. "Multi-threaded graph partitioning". In: *Proc. of the 27th IPDPS*, pages 225–236. 2013. [see pages 28–30, 49, 61, 67, 90]

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014. [see page 14]

[LK16] D. LaSalle and G. Karypis. “A parallel hill-climbing refinement algorithm for graph partitioning”. In: *Proc. of the 45th ICPP*, pages 236–241. 2016. [see pages 51, 61, 67]

[LMP15] M. von Looz, H. Meyerhenke, and R. Prutkin. “Generating Random Hyperbolic Graphs in Subquadratic Time”. In: *Proc. of the 26th ISAAC*, pages 467–478. 2015. [see pages 13, 14]

[Lub85] Michael Luby. “A Simple Parallel Algorithm for the Maximal Independent Set Problem”. In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 1–10. 1985. [see page 27]

[MB07] Fredrik Manne and Rob H. Bisseling. “A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem”. In: *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*, pages 708–717. 2007. [see pages 27, 30]

[MD97] Burkhard Monien and Ralf Diekmann. “A Local Graph Partitioning Heuristic Meeting Bisection Bounds”. In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, Hyatt Regency Minneapolis on Nicollet Mall Hotel, Minneapolis, Minnesota, USA, March 14-17, 1997*. 1997. [see page 46]

[Mey12] Henning Meyerhenke. “Shape optimizing load balancing for MPI-parallel adaptive numerical simulations”. In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*. Edited by David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Volume 588. Contemporary Mathematics, pages 67–82. ISBN: 978-0-8218-9038-7. American Mathematical Society, 2012. [see pages 55, 62, 67, 90]

[MH14] Fredrik Manne and Mahantesh Halappanavar. “New Effective Multithreaded Matching Algorithms”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 519–528. 2014. [see pages 31, 68]

[MMS09a] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. “A new diffusion-based multilevel algorithm for computing graph partitions”. In: *J. Parallel Distrib. Comput.* 69.9 (2009), pages 750–761. [see pages 48, 55, 58, 61, 67, 90, 109]

[MMS09b] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. “Graph partitioning and disturbed diffusion”. In: *Parallel Computing* 35.10-11 (2009), pages 544–569. [see pages 47, 48]

[Moo98] Gordon E Moore. “Cramming more components onto integrated circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pages 82–85. IEEE. [see page 3]

[MPD00] Burkhard Monien, Robert Preis, and Ralf Diekmann. “Quality matching and local improvement for multilevel graph-partitioning”. In: *Parallel Computing* 26.12 (2000), pages 1609–1634. [see pages 18, 46, 56, 57]

[MS04] Burkhard Monien and Stefan Schamberger. “Graph Partitioning with the Party Library: Helpful-Sets in Practice”. In: *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004), 27-29 October 2004, Foz do Iguacu, Brazil*, pages 198–205. 2004. [see pages 46, 47]

[MS07] Jens Maue and Peter Sanders. “Engineering Algorithms for Approximate Weighted Matching”. In: *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, pages 242–255. 2007. [see page 23]

[MS08] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008. [see pages 41, 89]

[MS12] Henning Meyerhenke and Thomas Sauerwald. “Beyond Good Partition Shapes: An Analysis of Diffusive Graph Partitioning”. In: *Algorithmica* 64.3 (2012), pages 329–361. [see pages 47, 48]

[MSD19] Tobias Maier, Peter Sanders, and Roman Dementiev. “Concurrent Hash Tables: Fast and General (?)!” In: *ACM Transactions on Parallel Computing (TOPC)* 5.4 (2019), page 16. ACM. [see pages 82, 85]

[MSS14] Henning Meyerhenke, Peter Sanders, and Christian Schulz. “Partitioning Complex Networks via Size-Constrained Clustering”. In: *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 351–363. 2014. [see pages 25, 45, 59, 66, 68, 70, 71, 90, 140, 141, 159]

[MSS17] Henning Meyerhenke, Peter Sanders, and Christian Schulz. “Parallel Graph Partitioning for Complex Networks”. In: *IEEE Trans. Parallel Distrib. Syst.* 28.9 (2017), pages 2625–2638. [see pages 33, 53, 61, 67, 68, 87, 90]

[Mur] *MurmurHash*. https://en.wikipedia.org/wiki/MurmurHash. [see page 178]

[MW47] Henry B Mann and Donald R Whitney. “On a test of whether one of two random variables is stochastically larger than the other”. In: *The annals of mathematical statistics* (1947), pages 50–60. JSTOR. [see page 15]

[Myg] URL: https://github.com/yarchi/KaHIP/tree/add_parallel_local_search. [see page 5]

[New06] M. E. J. Newman. “Modularity and community structure in networks”. In: 103.23 (2006), pages 8577–8582. National Academy of Sciences. [see page 31]

[Nor] *Wikipedia article: Normal distribution*. URL: https://en.wikipedia.org/wiki/Normal_distribution. [see page 73]

[NU13] Joel Nishimura and Johan Ugander. “Restreaming graph partitioning: simple versatile algorithms for advanced balancing”. In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 1106–1114. 2013. [see page 17]

[OS10] Vitaly Osipov and Peter Sanders. “n-Level Graph Partitioning”. In: *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, pages 278–289. 2010. [see pages 24, 44, 58, 72, 86]

[PB94] John R Pilkington and Scott B Baden. “Partitioning with spacefilling curves”. In: (1994). Citeseer. [see page 17]

[Pel07] François Pellegrini. “A Parallelisable Multi-level Banded Diffusion Scheme for Computing Balanced Partitions with Smooth Boundaries”. In: *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, pages 195–204. 2007. [see pages 48, 55, 58]

[Pel12] François Pellegrini. “Scotch and PT-scotch graph partitioning software: an overview”. In: *Combinatorial Scientific Computing* (2012), pages 373–406. Chapman and Hall/CRC. [see pages 60, 67, 90]

[Pet12] Seth Pettie. “A simple reduction from maximum weight matching to maximum cardinality matching”. In: *Inf. Process. Lett.* 112.23 (2012), pages 893–898. [see page 21]

[PM07] D. A. Papa and I. L. Markov. “Hypergraph Partitioning and Clustering”. In: *Handbook of Approximation Algorithms and Metaheuristics*. 2007. DOI: 10.1201/9781420010749.ch61. URL: http://dx.doi.org/10.1201/9781420010749.ch61. [see page 184]

[Pon+94] Ravi Ponnusamy, Nashat Mansour, Alok N. Choudhary, and Geoffrey Charles Fox. “Graph Contraction for Mapping Data on Parallel Computers: A Quality-Cost Tradeoff”. In: *Scientific Programming* 3.1 (1994), pages 73–82. [see pages 18, 56]

[Pre01] Robert Preis. “Analyses and design of efficient graph partitioning methods”. PhD thesis. Universitat, Paderborn, 2001. [see pages 18, 56]

[Pre99] Robert Preis. “Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs”. In: *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*, pages 259–269. 1999. [see pages 22, 27, 58, 68, 70]

[PS04] Seth Pettie and Peter Sanders. “A simpler linear time 2/3-epsilon approximation for maximum weight matching”. In: *Inf. Process. Lett.* 91.6 (2004), pages 271–276. [see page 23]

[PSL90] Alex Pothen, Horst D Simon, and Kang-Pu Liou. "Partitioning sparse matrices with eigenvectors of graphs". In: *SIAM journal on matrix analysis and applications* 11.3 (1990), pages 430–452. SIAM. [see page 57]

[PT11] M. Patrascu and M. Thorup. "The Power of Simple Tabulation Hashing". In: *Proceedings of the 43rd ACM STOC*, pages 1–10. ISBN: 978-1-4503-0691-1. 2011. [see page 88]

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: *Physical review E* 76.3 (2007), page 036106. APS. [see pages 4, 5, 25, 67, 140]

[RRW10] Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele. "Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations". In: *Math. Program.* 121.2 (2010), pages 307–335. [see page 35]

[San93] Laura A. Sanchis. "Multiple-Way Network Partitioning with Different Cost Functions". In: *IEEE Trans. Computers* 42.12 (1993), pages 1500–1504. [see page 42]

[San99] Peter Sanders. "Fast Priority Queues for Cached Memory". In: *ACM Journal of Experimental Algorithmics* 5 (1999), pages 312–327. [see pages 140, 144, 145]

[SB11] Richard S Sutton and Andrew G Barto. "Reinforcement learning: An introduction". In: (2011). Cambridge, MA: MIT Press. [see page 136]

[Sci] *SciPy: Probability Plot*. 2019. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.probplot.html. [see pages 73, 74]

[SDI06] Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. "Locality-Sensitive Hashing Using Stable Distributions". In: *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*. MITP, 2006. ISBN: 9780262256957. URL: https://ieeexplore.ieee.org/document/6282722. [see page 179]

[Sha+16] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. "Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks." In: *NSDI*, pages 455–468. 2016. [see page 65]

[Shu+13] J. Shun, G.E. Blelloch, J. T. Fineman, and P. B. Gibbons. "Reducing contention through priority updates". In: *Proc. of the 25th SPAA*, pages 152–163. 2013. [see page 86]

[Shu+16] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. "Parallel Local Graph Clustering". In: *PVLDB* 9.12 (2016), pages 1041–1052. [see page 136]

[Sim91] Horst D Simon. “Partitioning of unstructured problems for parallel processing”. In: *Computing systems in engineering* 2.2 (1991), pages 135–148. [see pages 17, 35, 56]

[SK12] Isabelle Stanton and Gabriel Kliot. “Streaming graph partitioning for large distributed graphs”. In: *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’12, Beijing, China, August 12-16, 2012*, pages 1222–1230. 2012. [see pages 17, 140]

[SKK03] Kirk Schloegel, George Karypis, and Vipin Kumar. “Sourcebook of Parallel Computing”. In: edited by Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. Chapter Graph Partitioning for High-performance Scientific Simulations, pages 491–541. ISBN: 1-55860-871-0. URL: http://dl.acm.org/citation.cfm?id=941480.941499. [see page 17]

[Slo+17] George M. Slota, Sivasankaran Rajamanickam, Karen D. Devine, and Kamesh Madduri. “Partitioning Trillion-Edge Graphs in Minutes”. In: *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 646–655. 2017. [see page 62]

[SM16] Christian L. Staudt and Henning Meyerhenke. “Engineering Parallel Algorithms for Community Detection in Massive Networks”. In: *IEEE Trans. Parallel Distrib. Syst.* 27.1 (2016), pages 171–184. [see pages 32, 33, 79, 136, 142]

[Smi82] Alan Jay Smith. “Cache memories”. In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pages 473–530. ACM. [see page 11]

[SMR14] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. “PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks”. In: *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014*, pages 481–490. 2014. [see pages 62, 68, 90]

[SN11] Jyothish Soman and Ankur Narang. “Fast Community Detection Algorithm with GPUs and Multicore Architectures”. In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 568–579. 2011. [see page 31]

[Sou35] R. V. Southwell. “Stress-Calculation in Frameworks by the Method of “Systematic Relaxation of Constraints””. In: *Proc. of the Royal Society of London* (1935), pages 56–95. The Royal Society. [see page 66]

[SS11] Peter Sanders and Christian Schulz. "Engineering Multilevel Graph Partitioning Algorithms". In: *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pages 469–480. 2011. [see pages 4, 43, 45, 47, 51, 59, 66–68, 71, 72, 90, 98]

[SS12a] Peter Sanders and Christian Schulz. "Distributed Evolutionary Graph Partitioning". In: *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*, pages 16–29. 2012. [see page 62]

[SS12b] Peter Sanders and Christian Schulz. "High quality graph partitioning." In: *Graph Partitioning and Graph Clustering* 588.1 (2012). [see page 47]

[SSP07] J. Singler, P. Sanders, and F. Putze. "MCSTL: The multi-core standard template library". In: *Proc. of the 13th Euro-Par* (2007), pages 682–694. [see pages 79, 82]

[SSS12] Ilya Safro, Peter Sanders, and Christian Schulz. "Advanced Coarsening Schemes for Graph Partitioning". In: *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*, pages 369–380. 2012. [see pages 24, 58, 66]

[ST97] Horst D. Simon and Shang-Hua Teng. "How Good is Recursive Bisection?" In: *SIAM J. Scientific Computing* 18.5 (1997), pages 1436–1445. [see page 36]

[Stü01] Klaus Stüben. "An introduction to algebraic multigrid". In: *Multigrid* (2001), pages 413–532. [see pages 48, 58]

[Sui+10] Xin Sui, Donald Nguyen, Martin Burtscher, and Keshav Pingali. "Parallel Graph Partitioning on Multicore Architectures". In: *Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*, pages 246–260. 2010. [see pages 62, 67]

[SW91] John E. Savage and Markus G. Wloka. "Parallelism in Graph-Partitioning". In: *J. Parallel Distrib. Comput.* 13.3 (1991), pages 257–272. [see pages 45, 64, 71]

[Tbb] *Intel Threading Building Blocks*. https://www.threadingbuildingblocks.org. [see pages 79, 80, 88]

[TOS00] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid*. Elsevier, 2000. [see pages 48, 58]

[Tso+14] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. "FENNEL: streaming graph partitioning for massive scale graphs". In: *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342. 2014. [see page 17]

[UB13] J. Ugander and L. Backstrom. “Balanced Label Propagation for Partitioning Massive Graphs”. In: *Proc. of 6th WSDM*, pages 507–516. 2013. [see pages 63, 67, 90]

[UC00] Ryuhei Uehara and Zhi-Zhong Chen. “Parallel approximation algorithms for maximum weighted matching in general graphs”. In: *Inf. Process. Lett.* 76.1-2 (2000), pages 13–17. [see page 27]

[VH05] Doratha E. Drake Vinkemeier and Stefan Hougardy. “A linear-time approximation algorithm for weighted matchings in graphs”. In: *ACM Trans. Algorithms* 1.1 (2005), pages 107–122. [see pages 22, 23]

[Vit01] Jeffrey Scott Vitter. “External memory algorithms and data structures”. In: *ACM Comput. Surv.* 33.2 (2001), pages 209–271. [see page 10]

[VN93] John Von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pages 27–75. IEEE. [see page 9]

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. “Algorithms for Parallel Memory I: Two-Level Memories”. In: *Algorithmica* 12.2/3 (1994), pages 110–147. [see page 10]

[WA] University of Milano Laboratory of Web Algorithms. *Datasets*. [see pages 3, 14, 15]

[Wal] Chris Walshaw. *The graph partitioning archive*. URL: http://chriswalshaw.co.uk/partition/. [see pages 58, 59, 61]

[Wal03] Chris Walshaw. “An exploration of multilevel combinatorial optimisation”. In: *Multilevel Optimization in VLSICAD*. Springer, 2003, pages 71–123. [see page 20]

[Wal04] Chris Walshaw. “Multilevel Refinement for Combinatorial Optimisation Problems”. In: *Annals OR* 131.1-4 (2004), pages 325–372. [see pages 18, 21, 56]

[WC00a] Chris Walshaw and Mark Cross. “Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm”. In: *SIAM J. Scientific Computing* 22.1 (2000), pages 63–80. [see pages 18, 46, 56, 57, 91, 159]

[WC00b] Chris Walshaw and Mark Cross. “Parallel optimisation algorithms for multilevel mesh partitioning”. In: *Parallel Computing* 26.12 (2000), pages 1635–1660. [see pages 53, 60]

[WC+02] C Walshaw, M Cross, et al. “Parallel mesh partitioning on distributed memory systems”. In: *Computational mechanics using high performance computing* (2002), pages 59–78. [see page 60]

[WC08] Chris Walshaw and Mark Cross. “JOSTLE: parallel multilevel graph-partitioning software – an overview”. In: 2008. [see page 60]

[WCE95] Chris Walshaw, Mark Cross, and Martin G. Everett. "A Localized Algorithm for Optimizing Unstructured Mesh Partitions". In: *IJHPCA* 9.4 (1995), pages 280–295. [see page 46]

[WCE97] Chris Walshaw, Mark Cross, and Martin G. Everett. "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes". In: *J. Parallel Distrib. Comput.* 47.2 (1997), pages 102–108. [see pages 29, 60]

[Wik19] Wikipedia contributors. *Q–Q plot — Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/w/index.php?title=Q%E2%80%93Q_plot&oldid=896576623. [see page 73]

[Wil45] Frank Wilcoxon. "Individual Comparisons by Ranking Methods". In: *Biometrics Bulletin* 1.6 (1945), pages 80–83. ISSN: 00994987. International Biometric Society. [see page 15]

[Wil91] Roy D. Williams. "Performance of dynamic load balancing algorithms for unstructured mesh calculations". In: *Concurrency - Practice and Experience* 3.5 (1991), pages 457–481. [see pages 17, 56]

[WW93] Dorothea Wagner and Frank Wagner. "Between Min Cut and Graph Bisection". In: *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*, pages 744–750. 1993. [see page 64]

[Wyl79] James C Wyllie. *The complexity of parallel computations*. Technical report. Cornell University, 1979. [see page 9]

[XN98] Cheng-Zhong Xu and Yibing Nie. "Relaxed Implementation of Spectral Methods for Graph Partitioning". In: *Solving Irregularly Structured Problems in Parallel, 5th International Symposium, IRREGULAR '98, Berkeley, California, USA, August 9-11, 1998, Proceedings*, pages 366–375. 1998. [see page 17]

[Zeh02] Norbert Zeh. "I/O-efficient Graph Algorithms". In: *EEF Summer School on Massive Data Sets* (2002). [see pages 5, 140, 144, 146]

[Zum12] Gerhard Zumbusch. *Parallel Multilevel Methods: Adaptive Mesh Refinement and Loadbalancing*. Springer Science & Business Media, 2012. [see page 17]

List of Publications

In Conference Proceedings

Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. "High-Quality Shared-Memory Graph Partitioning". In: *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 659–671. 2018

Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. "Engineering a direct k-way Hypergraph Partitioning Algorithm". In: *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017*. Pages 28–42. 2017

Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. "Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner". In: *PVLDB* 10.11 (2017), pages 1418–1429

Yaroslav Akhremtsev and Peter Sanders. "Fast Parallel Operations on Search Trees". In: *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*, pages 291–300. 2016

Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. "(Semi-)External Algorithms for Graph Partitioning and Clustering". In: *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pages 33–43. 2015

Theses

Yaroslav Akhremtsev. "Design and Development of Functional Programming Language". Bachelor Thesis. Institute of Automatics and Computer Engineering, Moscow Power Engineering Institute, Russia. May 2011

Yaroslav Akhremtsev. "Development and Analyzes of Shortest Paths Algorithms for Road Networks". Master Thesis. Institute of Automatics and Computer Engineering, Moscow Power Engineering Institute, Russia. May 2013