Karlsruhe Institute of Technology

Bachelor thesis

Engineering of Algorithms for Very Large k Partitioning

Manuel Haag

Date: October 1, 2021

Supervisors: Prof. Dr. Peter Sanders
M. Sc. Tobias Heuer
Prof. Dr. Christian Schulz
M. Sc. Daniel Seemaier

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 30. September 2021

Abstract

Graph partitioning is a NP-Complete [25] problem that asks to partition the node set of a graph into k blocks of "roughly equal" size while simultaneously minimizing the number of edges connecting the blocks. One of the most prominent applications of graph partitioning is parallel computing where we want to distribute a computational task to a compute node cluster such that each node receives an equal amount of work and the communication between the nodes required to the complete the task is minimized. However, with growing number of computing cores the problem of partitioning graphs into a large number of blocks becomes increasingly important. Recently, a shared-memory multilevel graph partitioning algorithm, called KaMinPar [19], was published that can partition graphs with billion of edges into million of blocks within minutes. However, the extreme case where each block contains only a constant number of nodes is relatively unexplored. In such scenarios, which we consider as *very* large k partitioning, the blocks of the partition should be formed by nodes that are tightly-coupled in the original graph. Therefore, we believe that for this case much simpler algorithms that are specifically tailored to very large k partitioning yield better results than traditional complex multilevel algorithms. To this end, we implement and evaluate a shared-memory partitioner that directly partitions the input graph into the desired number of blocks and afterwards, apply a novel refinement algorithm to improve the solution quality. For block sizes that only contain up to 8 nodes, our best configuration produces better solutions than KaMinPar on 70% of our benchmark instances, and is on average 4 times faster. Further, for block sizes up to 32 nodes, our algorithm produces partitions with comparable quality to KaMinPar and is still a factor of 1.5 faster.

Zusammenfassung

Graphpartitionierung ist ein NP-vollständiges [25] Problem, bei dem es darum geht, die Knotenmenge eines Graphen in k Blöcke "annähernd gleicher" Größe zu partitionieren und gleichzeitig die Anzahl der Kanten, die die Blöcke verbinden, zu minimieren. Eine der bekanntesten Anwendung von Graphpartitionierung ist paralleles Rechnen, bei dem man eine Berechnung so auf ein Cluster von Rechenknoten verteilt, dass jeder Knoten die gleiche Menge an Arbeit erhält und die Kommunikation zwischen den Knoten, die benötigt wird, um die Aufgabe zu erfüllen, minimiert wird. Jedoch mit wachsender Anzahl von Rechenkernen wird das Problem Graphen in große Blöcke zu partitionieren zunehmend wichtiger. Kürzlich wurde ein Shared-Memory Multilevel-Graphpartitionierungs-Algorithmus, namens KaMinPar [19], veröffentlicht, welcher Graphen mit Milliarden von Kanten innerhalb von Minuten in Millionen von Blöcken partitionieren kann. Jedoch der extreme Fall, in dem jeder Block nur eine konstante Anzahl von Knoten enthält, ist relativ unerforscht. In solchen Szenarien, welche wir als *sehr* große k Partitionierung betrachten, sollten die Blöcke der Partition aus Knoten gebildet werden, die eng miteinander gekoppelt sind. Daher glauben wir, dass für diesen Fall viel einfachere Algorithmen, die speziell für sehr großes k zugeschnitten sind, bessere Ergebnisse liefern als traditionelle komplexe Multilevel-Algorithmen. Zu diesem Zweck implementieren und evaluieren wir einen Shared-Memory Partitionierer, der direkt den Eingabegraphen in die gewünschte Anzahl von Blöcke partitioniert und anschließend einen neuen Verfeinerungsalgorithmus anwendet, um die Lösungsqualität zu verbessern. Für Blockgrößen, die nur bis zu 8 Knoten enthalten, produziert unsere beste Konfiguration bessere Lösungen als KaMinPar auf 70% unserer Benchmark-Instanzen, und ist im Durchschnitt viermal so schnell. Außerdem erzeugt unser Algorithmus für Blockgrößen bis zu 32 Knoten Partitionen mit vergleichbarer Qualität wie KaMinPar und ist immer noch um einen Faktor von 1.5 schneller.

Contents

1 Introduction

Graphs are a mathematical model for network-like structures. They appear in many places throughout computer science and have a wide range of applications also in other areas such as social sciences, biology, linguistics and physics. One very useful operation is *balanced graph partitioning* that asks to partition the node set of a graph into k blocks of "roughly equal" size, while cutting only few edges. Balanced graph partitioning is NP-Complete [25] and has no constant factor approximation on general graphs [8]. Thus in practice, one tries to find good solutions using heuristics.

A prominent application of graph partitioning is parallel computing. In scientific simulations, graph partitioning is used to map a computational network to processors, such that each processor has roughly an equal number of mesh elements and communication cost between the processors is minimized [41]. Existing research mostly focused on partitioning graphs into small numbers of blocks, $2 \leq k \leq 256$. However, with growing numbers of processors in parallel machines, we are interested in large values of k – in the order of millions. For example, the fastest supercomputer as of June 2021, Supercomputer Fugaku, has about 7.6 million compute cores [6]. Graph partitioning also finds application in accelerating routing algorithms [35], in Very Large Scale Integration (VLSI) [5], as first approximations for community detection algorithms [36] and in image segmentation [37].

The most successful approach used by many state-of-the-art partitioners is the *multilevel paradigm*. It consists of three phases: During *coarsening*, a graph hierarchy is build by successively contracting node sets computed by clustering or matching algorithms to obtain several coarser approximations of the input graph. When the number of nodes of the coarse graph falls below a certain threshold, a possibly expensive *initial partitioning* algorithm computes a partition into k blocks of the coarsest graph. In the last step the contractions are *uncoarsened* in reverse order, and, at each level, different *refinement* algorithms are used to improve the quality of the partition.

However, if k is large, the coarsest graph that is used for initial partitioning can be still large. As a consequence, many currently available multilevel frameworks compute either highly imbalanced solutions or have prohibitive running time for this case [19]. Recently, a shared-memory multilevel graph partitioning algorithm, called KaMinPar [19], was published, that is specifically tailored for partitioning graphs into a large number of blocks, which substantially mitigates these problems.

For very large k, i.e. $k \in \mathcal{O}(|V|)$, the blocks of the partition contain only a small number of nodes. In the extreme setting that every block is only allowed to contain two nodes, the problem is equivalent to the maximum weight matching problem, since minimizing the edge cut corresponds to maximizing the total edge weight between matched nodes. For block sizes larger than two, the problem is NP-Complete. Good partitions for this case contain many densely connected blocks. But since the block sizes are small, nodes within such blocks must have some locality. However, multilevel algorithms have a more global view on the partition of the graph, since they refine the partition in coarser as well as finer levels of the graph hierarchy. Therefore, we believe that there must be some trade-off depending on the number of nodes in a block, where simpler non-multilevel algorithms are comparable or considerably better than traditional complex multilevel algorithms.

1.1 Problem Statement

In this thesis, we want to design and evaluate algorithms to solve graph partitioning for the case that k is very large, i.e. $k \in \mathcal{O}(|V|)$ and each block contains only a constant number of

nodes. For this purpose, a framework should be developed that consists of the following two phases: *initial partitioning* and *refinement*. The initial partitioning phase should implement algorithms similar to techniques used in the initial partitioning phase of multilevel partitioners, such as random and graph growing approaches. For the refinement phase, different local search algorithms should be investigated. The main task of this work is an extensive experimental evaluation of the created algorithms on different benchmark instances. A major goal is to significantly reduce the running time compared to KaMinPar while producing solution of comparable or better quality on most benchmark instances.

1.2 Contribution

In this work, we implemented and evaluated four different initial partitioning and refinement algorithms. We parallelized our most successful algorithms and compared them to KaMinPar where each block is only allowed to a small number of nodes. Using simple graph growing techniques for initial partitioning and a novel refinement algorithm, we outperform KaMinPar for block sizes up to 8 nodes on 70% of the instances, while being faster by a factor of 4. For block sizes up to 32, our algorithm is faster than KaMinPar by a factor of 1.5 with comparable quality. Simple Label Propagation (LP) for refinement, outperforms KaMinPar with less quality than our algorithm for block sizes up to 4 and is comparable for block sizes up to 16. However, LP due to its simplicity is an order of magnitude faster (by a factor of 29) than KaMinPar.

1.3 Outline

In Section 2 we introduce basic definitions and required notation. We summarize related work in the area of graph partitioning in Section 3, where we mostly focus on the stages of the multilevel paradigm. Then, we describe our very large k partitioning framework and present the experimental evaluation of our algorithms in Sections 4 and 5. Section 6 summarizes the results and outlines future work.

2 Preliminaries

2.1 Basic Definitions

Let V be a finite set and d an integer. The set $\binom{V}{d} := \{S \subseteq V \mid |S| = d\}$ is the set of all subsets of V of size d. An undirected weighted *graph* G is a tuple (V, E, c, ω), where V is the set of *nodes*, $E \subseteq \binom{V}{2}$ the set of *edges*, with node weights $c : V \to \mathbb{R}_{>0}$ and edge weights $\omega : E \to \mathbb{R}_{>0}$. Let $N(v) := \{w \in V \mid \{w, v\} \in E\}$ denote the *neighborhood* of v and $E(A, B) := \{\{v, w\} \in E \mid v \in A, w \in B\}$ the set of *cut edges* between the node sets $A, B \subseteq V$. For compact notation define $E(v, A) = E(A, v) := E(\{v\}, A)$ for $v \in V$, $A \subseteq V$. The *degree* of a node v is the number of adjacent neighbors $d(v) := |N(v)|$. The *maximum degree* of the graph is denoted by $\Delta := \max_{v \in V} d(v)$. The definitions of $N(v)$, c, ω are extended in the intuitive way to sets. For $V' \subseteq V$, $E' \subseteq E$ define $N(V') := \bigcup_{v \in V'} N(v) \setminus V'$, $c(V') := \sum_{v \in V'} c(v)$, $\omega(E') := \sum_{e \in E'} \omega(e)$.

A *path* P in a graph G is a sequence of nodes $v_1, v_2, \ldots, v_l \in V$, where nodes that are adjacent in the sequence are also adjacent in G, i.e. $\{v_i, v_{i+1}\} \in E$ for $i \in \{1, 2, \ldots, l-1\}$. A graph is called *connected*, if for each pair of nodes $v, w \in V$, there exists a path from v to w. The *length* of a path is the number of edges in P. Furthermore, the *distance* $d(v, w)$ between two nodes

$v, w \in V$ is the length of the shortest path between v and w, if there is any, otherwise we set $d(v, w) := \infty$.

A *matching* $M \subseteq E$ is a set of edges that do not share any common nodes, i.e., the graph (V, M) has maximum degree one. The weight of a matching is $\omega(M)$. A matching is called *maximal*, if M can not be extended by adding one more edge $e \in E \setminus M$. A *maximum weight* matching is a matching maximizing $\omega(M)$. A matching M is called a *perfect*, if every node is incident to an edge of the matching.

Contracting an edge $\{u, v\}$ means to replace the nodes u and v by a new node x and connect it to the former neighbors of u, v. We set $c(x) = c(u) + c(v)$ as the new node weight. If u, v have a shared neighbor w, we insert only one edge $\{x, w\}$ with new edge weight $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$. Otherwise if w is a neighbor of exactly one of u, v, say u, the edge $\{x, w\}$ with edge weight $\omega(\{x, w\}) = \omega(\{u, w\})$ is inserted. Uncontracting an edge e undoes its contraction. Note that contraction and uncontraction preserve the total node weight $c(V)$.

2.2 Balanced Graph Partitioning

Given a number $k \in \mathbb{N}_{>1}$ and node sets $V_1, V_2, \ldots, V_k \subseteq V$ with the properties:

$$\begin{aligned} a) \quad & V_1 \cup V_2 \cup \cdots \cup V_k = V \\ b) \quad & V_i \cap V_j = \emptyset \text{ for } i \neq j \end{aligned}$$

then $\{V_1, V_2, \ldots, V_k\}$ is called a k-way partition of G.

Given a fixed k the *balanced graph partitioning* problem is to find a k-way partition $\{V_1, V_2, \ldots, V_k\}$ of G minimizing the total cut $\sum_{i<j} E(V_i, V_j)$ under a given *balance constraint*. The *balance constraint* demands that all blocks have about the same size. More precisely $\forall i \in 1, 2, \ldots, k$: $c(V_i) \leq (1 + \epsilon)\frac{c(V)}{k} + \max_{v \in V} c(v)$ for some *imbalance* parameter $\epsilon \in \mathbb{R}_{\geq 0}$. The last term in this equation arises because each node is atomic and therefore a deviation of the heaviest node has to be allowed.

For the special case of this thesis, where the average block size $\frac{c(V)}{k}$ is constant, we will use a deviation by a fixed node weight $b \in \mathbb{N}_0$ as balance constraint $L_{max} := \frac{c(V)}{k} + b + \max_{v \in V}$. For our purposes, this notion of balance is more meaningful and is easier to control. A block V_i is *underloaded* if $V_i < L_{max}$ and *overloaded* if $V_i > L_{max}$. Recall that balanced graph partitioning is NP-Complete for every $k = \frac{|V|}{C}, C \geq 3$ [25] and has no constant factor approximation on general graphs [8].

Changing the block of a node v is called a *move*. The *gain* of a move from block A to block B is the number $g_B(v) := \omega(E(v, V_B)) - \omega(E(v, V_A))$. This number tells us how the total cut changes by this move. If the $g_B(v) \geq 0$, the cut decreases by $g_B(v)$ and the move is called *positive*, if $g_B(v) < 0$ the cut increases by $-g_B(v)$ and the move is called *negative*. A move with $g_B(v) = 0$ is also called *zero gain move*. A *max gain move* for a fixed node v is the move maximizing $g_B(v)$ for $B \in 1, 2, \ldots, k$.

3 Related Work

In this section, we will discuss related work. We mostly focus on the widely used *multilevel paradigm* described in Section 3.1 and outline different techniques used in the stages of the *multilevel paradigm*, namely *coarsening* (Section 3.2), *initial partitioning* (Section 3.3) and *refinement* (Section 3.4). Then follows a brief description of a geometric partitioning technique

using *spacefilling curves* Section 3.5. At the end we describe a partitioner *KaMinPar* Section 3.6, which uses a novel technique called *Deep Multilevel Graph Partitioning* based on the multilevel paradigm.

3.1 Multilevel Paradigm

The multilevel paradigm [20] is successfully used in practice by many state-of-the-art partitioners. In the *direct k-way partitioning* variant, the algorithm has three phases (see Figure 1). First, a hierarchy of graphs is build by successively *coarsening* the graph. This is achieved by contracting node sets, computed by a clustering or matching algorithm, until it reaches a predefined size in $\mathcal{O}(k)$ or the procedure converges. Each graph is thereby a smaller approximation of the graph from the previous level. Then, a possibly expensive *inital partitioning algorithm* computes a partition of the coarsest graph. By the way contraction is defined, this partition corresponds to a valid partition in every level of the graph hierarchy. In the *uncoarsening phase* the contractions are subsequently reverted, the partition of the coarser graph is projected to the next graph in the hierarchy and then is improved by a *refinement algorithm*.

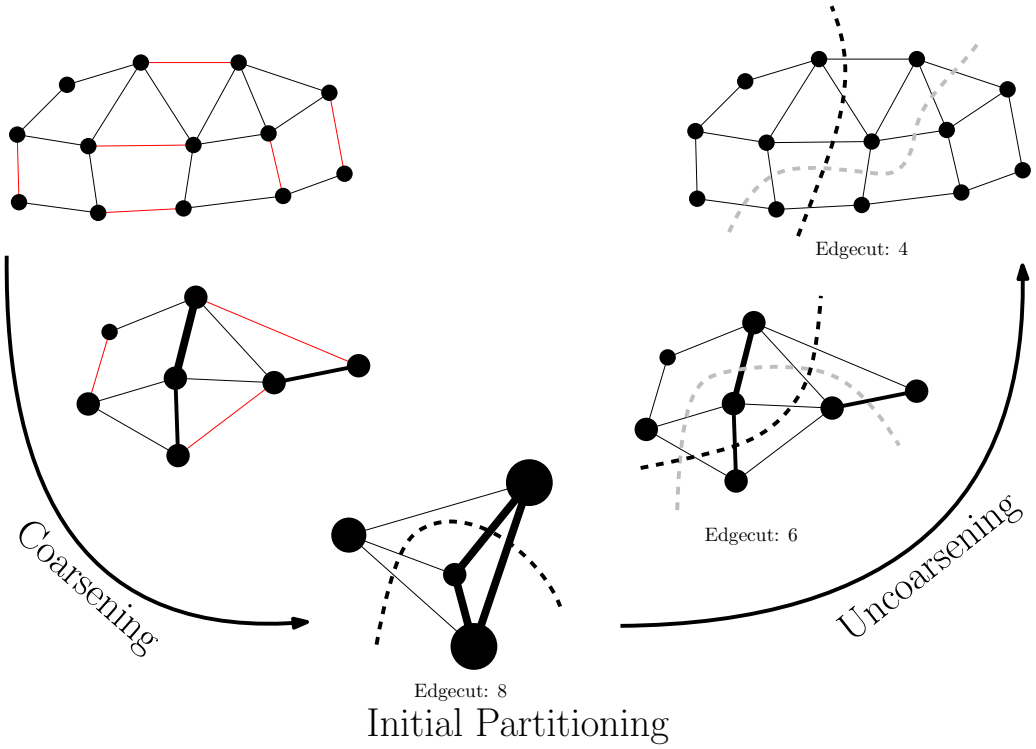


Figure 1: Graph Partitioning with Multilevel Paradigm.

3.2 Coarsening

The *coarsening phase* is the first stage in the multilevel paradigm, in which the hierarchy of smaller graphs is build. There are two important goals of coarsening [42]. Firstly, the contraction should quickly reduce the input size. Secondly, each computed level should reflect the global structure of the input network. Usually, clustering and matching algorithms are used for coarsening. In this section, we outline one clustering method based on Label Propagation as well as different matching algorithms used for finding good matching in a practical amount of time.

3.2.1 Size Constrained Label Propagation

Label Propagtion was originally proposed by Raghavan et al. to detect community structures in networks [39]. The original algorithm works as follows:

Initially, we assign to each node an unique label. The algorithm then works in rounds. In one round the nodes are traversed in some order. If we visit a node v, it is moved to the neighboring block V_i that the most neighbors are part of, i.e. to the cluster maximizing $|N(v) \cap V_i|$, with ties broken uniformly randomly. This process repeats until none of the vertices changed its label.

Meyerhenke et al. [33] utilized the Label Propagation algorithm to create a multilevel hierarchy. In one contraction, step nodes with the same label are contracted into one node. In contrast to the original algorithm, blocks of the cluster must fulfill a size constraint. If a block would exceed L_{max}, it would be impossible to find a partition of the contracted graph that fulfills the balance constraint. For this reason the authors introduced an upper bound $U := \max(\max_{v \in V} c(v), \frac{L_{max}}{f})$ for the maximal size of a block, where f is a tuning parameter.

A neighboring block V_i of a node v is called *eligible*, if $c(V_i)$ does not exceed U once v is moved to V_i. When a node v is visited, it is moved to the eligible block having the most neighbor of v. Additionally, they stop the algorithm after l iterations or if less than five percent of nodes changed their label.

3.2.2 Matching Based Coarsening

Matching-based coarsening techniques contract a large matching in every contraction step. This method can make use of existing matching algorithms. Optimal matchings can be computed in polynomial time but are in general too slow [13]. In practice heuristic matching algorithms are fast, produce good matchings and also can give an approximation guarantee [13]. We shortly outline some methods.

Random Matching The random maximal matching algorithm works as follows [4]: The nodes are visited in random order. If a node v is not matched, a random unmatched neighbor w of v is selected. If such w exists, the edge $\{v, w\}$ is added to the matching and v and w are marked as matched. Otherwise, v has no unmatched neighbor and v remains unmatched. The complexity of the described algorithm is $\mathcal{O}(|E|)$.

Greedy Edge Matching We can obtain a 2-approximation by always matching the heaviest unmatched edge in the graph [26]. This can be achieved by sorting the edges by edge weight and then scanning the edges in reverse order. Consider an edge e chosen by the greedy approach. If e is not in an optimal matching, then at most both endpoints of e are matched with other edges e_1, e_2. By the sorting $\omega(e_1) \leq \omega(e)$ and $\omega(e_2) \leq \omega(e)$. In total the optimal choice is not better than $2 \cdot \omega(e)$. The runtime $\mathcal{O}(|E| \log |E|)$ is dominated by the sorting algorithm.

Heavy Edge Matching The Heavy Edge Matching Algorithm is a simple $\mathcal{O}(|E|)$ matching algorithm [4]. The nodes are traversed in some order. Every unmatched node v is matched with the unmatched neighbor $w \in N(v)$ connected via the heaviest edge. This method is a more local version of the greedy approach and is also faster in practice. However, this algorithm has no approximation guarantees. Consider a triangle with edge weights 1,1, α, $\alpha > 1$. If the node with the two smaller edges is visited first, the optimal can not be taken anymore. Since α can be arbitrarily large, the algorithm has no approximation guarantees.

Path Growing Matching The Path Growing Algorithm improves the idea of the Heavy Edge Matching to obtain a 2-approximation in $\mathcal{O}(|E|)$ time. Starting from some node it constructs a path following the heaviest edge to an unassigned neighbor. The edges of the path are assigned in alternating fashion to two matchings M_1, M_2. This process is repeated until no edge can be matched anymore. Returning $\arg\max(\omega(M_1), \omega(M_2))$ yields a 2-approximation [26].

Global Paths Algorithm (GPA) GPA is a more sophisticated heuristic to obtain a 2-approxmiation in $\mathcal{O}(|E| \log |E|)$ time [32], which empirically gives considerably better results than other approximations. It finds a collection of paths and even length cycles and calculates optimal matchings for each paths and cycles using dynamic programming.

3.3 Initial Partitioning

The *initial partitioning phase* is the second phase of the multilevel paradigm. In this phase a k-way partition of the coarsest graph is computed. Since this partition is propagated to the top level, it has a major impact on the solution quality. The assumption in the multilevel paradigm is that the coarsest graph is small, so an expensive partitioning algorithm can be used. We describe two approaches. First, BFS-based and greedy graph growing, that grow a block around seed nodes. Secondly, we outline the idea of spectral methods, that use the information about connectivity in the second eigenvector of the Laplace-matrix to partition the graph.

3.3.1 Graph Growing

In general, graph growing algorithms describe a family of algorithms, which can be either BFS-based or based on greedy growing techniques. An algorithm to obtain a bipartition works as follows [17, 18]. A BFS traversal is initialized with a random start node. All the nodes touched by this BFS are assigned to block V_1. The BFS continues until half of the nodes were assigned to V_1. The remaining nodes $V \setminus V_1$ form the second block V_2. This method is very sensitive to the selection of the start node, so it is repeated multiple times with different start nodes.

Alternatively, one can also select a *peripheral node* as start node, which is a node x that maximizes the *eccentricity* $l(x) := \max_{y \in V} d(x, y)$ [17]. Such a node has maximum distance to a node out of all nodes and may be a good start node for computing a bipartition. Since finding a peripheral node is expensive, the authors describe a heuristic algorithm to obtain a node with high eccentricity $l(x)$, called a *pseudo peripheral node*. The algorithm works as follows: A BFS is started from a random node and the last node touched by BFS is chosen as a *pseudo peripheral node*. To get better results, one can repeat this process several times with the last found node and store the node x with maximal $l(x)$ as final start node.

A greedy variation of graph growing always adds the node to the block having the least increase in cut, instead of selecting the nodes in normal BFS-order [27]. These approaches can be extended intuitively to a k-way partition algorithm by using multiple BFS traversal and stop each after it has the average number of nodes $\frac{c(V)}{k}$ in the block. In Section 4.1.2 we describe a more detailed implementation of the k-way approach similiar to [23].

3.3.2 Spectral Partitioning

The spectral bisection method, which was first introduced by Donath and Hoffman [12], partitions an unweighted graph by calculating the second eigenvector v_2 of the Laplace-matrix

$L := D - A$, also known as the *Fiedler Vector*, where D is a diagonal matrix containing the node degrees and A is the adjacency matrix. A bisection is encoded in a vector by setting the entries of the respective nodes to 1 if they are in V_1 and -1 if they are in V_2. The key observation is that the number of cut edges $|C|$ can be expressed in the scalar $x^T Lx = \sum_{\{u,v\} \in E} (x_u - x_v)^2 = 4|C|$, where the vector x encodes the bisection. So the bisection problem can be reformulated as the solution to the optimization problem

$$\min\{x^T Lx \mid x^T 1 = 0, x^T x = n, x \in \{-1, 1\}^n\}$$

The objective function corresponds to the minimization of the edge cut and the first condition corresponds the constraint of perfect balance (assuming even n). It can be shown that if the integrality constraint is dropped, the Fiedler Vector v_2 is an optimal solution to the relaxed problem. In order to obtain a partition from a solution x to the relaxed problem, the median m of the entries of x is calculated. A node v with $x_v \leq m$ is assigned to V_1, otherwise it is assigned to V_2. There are generalizations for arbitrary k [9, 21], however, the case of very large k can not be handled efficiently and is still an issue [9].

3.4 Refinement

During the *uncoarsening phase*, the last phase of the multilevel paradigm, the contraction of the graph hierarchy is successively undone, and *refinement algorithms* are used to improve the solution quality after each uncontraction. We outline the use of Size Constrained Label Propagation for refinement, the Kernighan-Lin algorithm, the Fiduccia-Mattheyses algorithm and a flow-based refinement method.

3.4.1 Size Constrained Label Propagation

The Size Constrained Label Propagation algorithm for coarsening mentioned in Section 3.2.1 can also be used for refinement by setting the tuning parameter $f = 1$ and initially give each node a label representing its block rather than unique labels. This yields a simple and fast local search algorithm [33].

3.4.2 Kernighan-Lin

Kernighan-Lin [28], referred to as KL, is a local search algorithm to improve the edge cut of a given balanced bisection (V_1, V_2) while still maintaining the balance. The algorithm repeatedly searches node sets $A \subseteq V_1$, $B \subseteq V_2$ with $|A| = |B|$, such that swapping the respective blocks of nodes in A and B improves the cut until no further improvement can be found. One *pass* consists of finding and exchanging these sets. A pass works as follows. The algorithm searches for pairs of nodes $v \in V_1$, $w \in V_2$ to exchange blocks. Moving v to V_2 and w to V_1 has a total gain of $g(v, w) := g_2(v) + g_1(w)$ if v and w are adjacent and $g(v, w) := g_2(v) + g_1(w) - 2\omega(\{v, w\})$ otherwise, since the edge $\{v, w\}$ will still be in the cut after the swap. In one pass a node is only allowed to move once. In one *round* the two unmarked nodes $a_i \in V_1$, $b_i \in V_2$ maximizing $g(a_i, b_i)$ are exchanged and marked. Note that $g(a_i, b_i)$ can be negative. This procedure is repeated $p := \min(|V_1|, |V_2|)$ times. In the end the best prefix of swaps is applied to the graph, i.e. $l \leq p$ is set to the smallest index maximizing $\sum_{i=1}^{l} g(a_i, b_i)$ and $A := \bigcup_{i=1}^{l} \{a_i\}$, $B := \bigcup_{i=1}^{l} \{b_i\}$. The major drawback of KL is the expensive asymptotic runtime. The original implementation has $O(n^2 \log n)$ running time [28] and could be improved to $O(m \max(\log n, \Delta)$ where Δ denotes the maximum degree [14]. An advantage of the algorithm is that it can climb out of local minima to a certain extent by the way A and B are constructed.

3.4.3 Fiduccia-Mattheyses

The Fiduccia-Mattheyses, referred to as FM, algorithm is similar to KL algorithm, but has an improved asymptotic runtime of $\mathcal{O}(m)$ [15]. The major difference to KL is that instead of performing node swaps, the algorithm tries to perform node moves that satisfy the balance constraint. In one *pass* the FM algorithm alternating selects one block and moves the node with the highest gain to the other block. During one pass a node is moved at most once. If no move can be found anymore, the algorithms performs a rollback to best seen solution during a pass. The procedure is repeated until no improvement can be achieved. This change allows a more efficient implementation using a data structure called bucket queue.

3.4.4 Flow Based Refinement

Sanders and Schulz [40] describe a flow-based algorithm to improve the edge cut of a given bipartition. The algorithm constructs a flow problem in an area around the boundary between the blocks. The area is chosen such that every s-t-cut induces a balanced cut in the original graph. The source and sinks are configured in such a way that a corresponding max-flow-min-cut algorithm induces a cut in the original graph, which improves the solution quality. The authors describe multiple extensions to the algorithm. For example, one can apply this method iteratively, search in larger areas for feasible cuts or apply the so-called most-balanced-minimum-cut heuristic to obtain better balanced min cuts [24]. The algorithm can be generalized to k-way partitioning by applying the method successively to pairs of adjacent blocks.

3.5 Geometric Partitioning

If a graph has coordinates for each node in space, one can use this geometric information to perform partitioning. Such graphs arise for example in scientific computing in finite element models or other geometrically-defined graphs. One approach uses spacefilling curves to partition the graph [38]. A spacefilling curve is a family of curves defined recursively in self similar manners. On each level the curve approximates the space more fine granular and can come arbitrarily close to any point. These curves preserve the spatial locality of the higher dimensional space, i.e. points that are close in the high dimensional space are close on the 1-dimensional curve. An example is the 2-dimensional Hilbert curve [3]. The idea of spacefilling curve partitioning is to map the nodes to a spacefilling curve, which reduces the d-dimensional partitioning problem to a 1-dimensional problem. This problem can be solved in linear time, by dividing the line into k intervals, such that all intervals have about the same node weight. An advantage of this method is that if subsequent partitions are needed, in which the nodes only moved a little, partitions can be generated by moving the boundaries between the intervals of the previous partition.

3.6 KaMinPar

KaMinPar is a graph partitioner that uses a novel approach based on the multilevel paradigm called *Deep Multilevel Graph Partitioning* [19]. Deep MGP continues coarsening deep into the initial partitioning phase to a size of $2C$, for a parameter C. Let $\text{ceil}_2(x)$ be x rounded up to the next power of two. The algorithm maintains the invariant that the partition of a coarse graph with n' nodes in the graph hierarchy has $k' = \min\{k, \text{ceil}_2(\frac{n'}{C})\}$ blocks. The choice of k' ensures that a bipartition algorithm works on roughly $2C$ nodes.

In the following we describe how a k-way partition is obtained. The graph hierarchy is unrolled level by level. Uncoarsening of one level works as follows: First, the partition of the lower level is projected to the uncoarsened graph and blocks are subdivided using bipartition algorithms, until the coarse graph has k' blocks. Possible violations of the balance constraint are repaired by applying balancing algorithms. Then k-way refinement algorithms are used to improve the solution quality. On the last level, if $k' < k$, blocks are further subdivided into k blocks to obtain a k-way partition. Figure 2 illustrates the procedure.

The Deep MGP can be seen as a hybrid approach between direct k-way partitioning and recursive bipartitioning: Like the direct k-way approach, Deep MGP coarsens and uncoarsens the graph only once and uses k-way local improvement algorithms throughout the graph hierarchy. Moreover, it enforces that (possibly expensive) bipartitioning algorithms are only applied to small graphs. They authors show in their evaluation that KaMinPar is an order of magnitude faster than other graph partitioners if k is large, while producing comparable solutions [19].

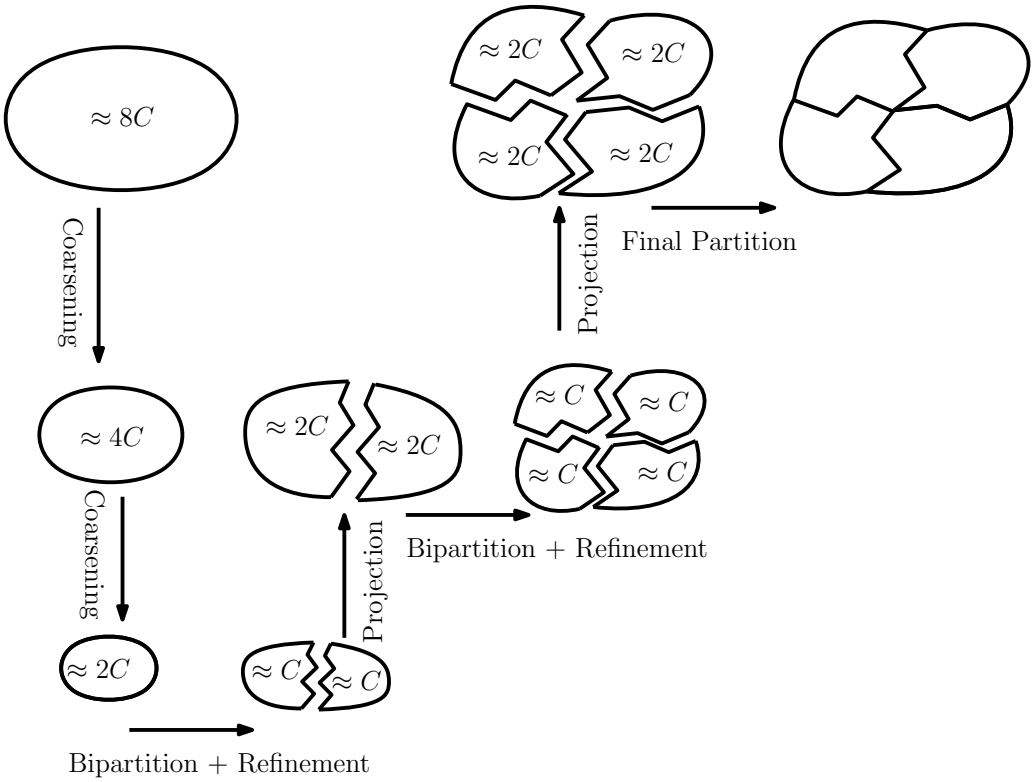


Figure 2: Illustration of Deep Multilevel Graph Partitioning.

4 A Partitioning Framework for Very Large k

In this section, we present our framework for very large k-way partitioning. By *very large* k we understand $k \in \mathcal{O}(|V|)$. In this setting, each block will contain only a small number of nodes. For very large k the block sizes are very small, so the partition should be formed by nodes that are tightly-coupled in the original graph. That is why we believe that there must be some trade-off depending on the number of nodes in a block, where simpler non-multilevel algorithms are comparable or considerably better than multilevel algorithms. For this purpose, we want to develop a framework that consists of the following two phases: *initial partitioning* and *refinement*. We deviate from the multilevel paradigm, since for very large k the coarsest graph is still very large, which is usually assumed to be very small. Our approach starts directly

in the *initial partitioning phase* (Section 4.1) and then continues to improve the partition using a *refinement algorithm* (Section 4.2).

Based on preliminary experiments we decided to parallelize our most promising algorithms. Thus, we describe a parallel version for the BFS initial partitioner and the refinement algorithms Label Propagation and Forward Path Refiner.

4.1 Initial Partitioning

The goal of the *initial partitioning phase* is to obtain a balanced k-way partition of the graph, which enables us to run a local search algorithm on the computed partition. We did not implement spectral methods (Section 3.3.2), as they are too slow for large k. Geometric methods, such as spacefilling curves (Section 3.5), need coordinates for each node and thus can not be used in the general setting.

We implement a partitioner that randomly assigns nodes to blocks, a partitioner based on graph growing similar to [23], as well as a partitioner inspired by matching-based coarsening outlined in Section 3.2.2. The random approach shows us how good the other techniques are performing compared to a random partition. Graph growing is a well-established technique in many initial partitioning portfolios such as Metis [27], KaMinPar [19], that is easy to implement and offers a good time-quality trade-off. Last, the matching approach with an optimal matching algorithm yields optimal solutions for $k = \frac{n}{2}$. This motivates the use of it for very-large k, where the graph is only contracted a few times.

4.1.1 Random Partitioning

The simplest way to compute a feasible k-way-partition is to assign the nodes one by one to a random block with respect to L_{max}. This technique will not result in good cuts, but it is more of a starting point of a partitioner for comparison. For the implementation, we use one array that contains all block IDs. A random block is selected by generating a random index in the range of the array and returning the block ID at that index. If after the movement of the node the block is full, we swap the block ID to the end of the array and decrease the array size by one.

4.1.2 Initial Partitioning based on Graph Growing Techniques

Based on the graph growing techniques in Section 3.3.1, we implemented the Breath First Search (BFS) partitioner. The idea is to grow each block by one separate BFS traversal. For this reason we define the procedure $BFS(u, k', L)$, that takes an unassigned start node u and runs a BFS on the unassigned nodes. Visited nodes are moved to $V_{k'}$ until the block weight of $V_{k'}$ is at least L and the BFS is stopped. If the BFS queue Q is empty, but $c(V_{k'}) < L$, a new unassigned node is pushed to Q in order to restart the BFS. In the following we call the node u for the very first BFS run *start node* and every other node u or node that is used to restart a BFS *next node*. We first outline how we set the block size L and then explain different start node and next node selection strategies.

Configuring the Block Sizes. We could run each BFS with the block size $L = L_{max}$, but then we would end up only with either full or empty blocks. As a consequence, many node swaps between block would be infeasible due to the balance constraint. Moreover, the partition would contain empty blocks, which can not be used by local search algorithms that only move

nodes between adjacent blocks. Hence, we want the block sizes to have approximately average weight $\frac{c(V)}{k}$. For simplicity, we assume that G is unweighted and let $r := |V| \mod \lfloor \frac{|V|}{k} \rfloor$. Then we can ensure that the block sizes differ by at most one by building r blocks of size $\lceil \frac{|V|}{k} \rceil$ and $k - r$ blocks of size $\lfloor \frac{|V|}{k} \rfloor$. In the weighted case one can set $L = \frac{c(V)}{k}$, but empty blocks can not be avoided, if many blocks exceed the average node weight $\frac{c(V)}{k}$ due to large nodes.
Now we describe the two strategies for the selection of the start node and the next node in more detail. In both cases we compare a simple strategy using randomization with a more sophisticated approach. For start node the latter is the *pseudo peripheral node* and for the next node the latter is the *border node*, which is a node that is adjacent to some block of the partition. Figure 3 shows an example of border nodes.

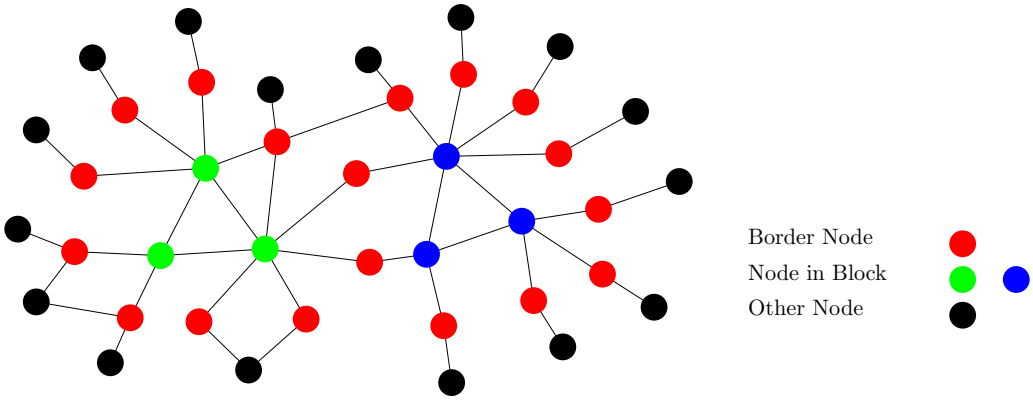


Figure 3: Example of border nodes.

Our experimental study indicates that the start node has a smaller impact on the behavior of the algorithm than the next node, which we select after we start a new BFS or restart an incomplete BFS. Choosing this node carefully can have a large impact on the solution quality.

Start Node Strategies. The first strategy is to simply select a random node $v \in V$. Since no nodes are assigned yet, we do not have to use an extra data structure to ensure that v is not marked. It is a simple an easy to implement strategy and has no further overhead.
The second strategy selects a *pseudo peripheral node*. In Section 3.3.1 we already explained the pseudo peripheral node and an algorithm to find one. Using a pseudo peripheral start node allows the blocks to grow from the outer part of the graph, which can lead to less fragmentation of the blocks.

Next Node Strategies. The *random next node* strategy selects a random node, which is not yet assigned to a block. We use two arrays A_1, A_2 to keep track of the unassigned nodes during the algorithm. A_1 stores the node IDs and A_2 the index of a node (starting at 1) to A_1, i.e. A_2 is the inverse permutation of A_1. In the beginning $A_1 = A_2 = [1, 2, \ldots, n]$. A random node is selected by returning the node ID of a random index in the range of the A_1. After the BFS marked a node, the node is removed from A_1 by looking up its index in A_2, swapping it with the last element of A_1 and decreasing the size of A_1 by one. The indices of the A_2 must be updated accordingly. Since every node is removed only once and since maximal $|V|$ times a random next nodes is needed, the running time is linear with $\mathcal{O}(|V|)$ extra space for the two arrays.
A disadvantage of the random next node strategy is that the blocks cover the graph unevenly. This leads to small holes of unassigned nodes during the algorithm. These nodes do not fill blocks completely and thus create fragmentation of blocks.

The second strategy tries to prevent a high fragmentation of blocks by selecting a border node as next node. In this way new blocks are adjacent to each other and less fragmentation of blocks is created. This can be efficiently implemented using a second queue Q'. Every time we insert a node into the BFS queue Q, we push this node also to Q', since the node is adjacent to the block grown by the current BFS. To avoid duplicate nodes from multiple BFS traversals in Q', we also mark the queued node in a bitset as processed and do not reinsert, if it is already processed. A border node can be found by popping nodes from Q' until we find an unmarked node. During the time a node is in Q', it could be visited by some BFS. In this setting, if the graph is connected, there always exists an unmarked border node in Q'. Consider a path between a marked node v and an unmarked node w. If we follow the nodes of the path from v to w, the first unmarked node z that we visit is a node that is adjacent to a marked node and thus $z \in Q'$. Because of the bitset, each node is only pushed and popped once, so the cost of finding one border node is amortized $\mathcal{O}(1)$. We will show in our evaluations, that this method performs significantly better than the random next node strategy.

Parallelization. In the parallel implementation, each thread runs a separate BFS on the unassigned nodes to construct a block. To avoid the sequential process of selecting pseudo peripheral nodes per thread, each thread starts the first BFS from a random node. Due to the better quality, we picked the border next node strategy over the random next node strategy. Each thread has a local queue Q for BFS and a local queue Q' for the border nodes. A global array A of boolean indicates at $A[v]$ if a node v was already assigned to a block. If a thread visits an unassigned node v and has not enough nodes in Q to fill the current block to at least L nodes, the thread tries to obtain the node by an atomic compare-and-swap operation on $A[v]$ and if successful, the node is pushed to Q. Otherwise, if the thread has enough nodes, the node is instead pushed to Q' for future border nodes without securing it by setting the entry $A[v]$. Opposed to the sequential description, we do not use a bitset to avoid duplicate nodes in Q', since that would cost $|V|$ entries per thread, which is bad for cache efficiency. A border node is obtained from Q' by popping nodes from Q' until an unassigned node was obtained by an atomic compare-and-swap operation. In contrast to the sequential implementation, we can not guarantee to get a border node, since other threads could have already assigned all nodes in Q'. In this case we employ the random next node strategy. We use a randomly shuffled global array of node IDs and each thread has a random start index i to that array. If a thread request a random next node, it scans the array starting from i, restarting at index 0, if the boundary is reached, until it obtained an unassigned node by compare-and-swap. The current index is stored in i for the next scan. So, one thread traverse the global array only once.

Greedy Graph Growing The BFS partitioner can be modified to a greedy graph growing algorithm, analogous to 3.3.1. We always select the node, that yields the smallest increase in the edge cut, i.e. the node with maximal number of edges to the current block. To achieve this we use a priority queue instead of the queue, which uses the number of edges to the current block as key with higher number having higher priority. However, the use of a priority queue comes at the cost of runtime, since removing an element of a priority queue is logarithmic in the size of the priority queue. Additionally, after a node v is moved to the current block, the keys of all the neighbors of v that are not assigned to a block yet must be updated, which also takes logarithmic time in the size of the priority queue for each neighbor.

4.1.3 Matching Contraction

Note that this approach works only for $k = n/2^l$, i.e. the block sizes are a power of two, and unweighted nodes (general case open for future work). The matching contraction approach is similar to the contraction phase in the multilevel paradigm, with the difference that we enforce uniform node weights and halving of the graph size. For this, we drop the constraint that matched nodes must be adjacent and extend a maximal matching by matching the remaining nodes. This way, the node weights double after one contraction except one node, if the number of nodes is odd. That is why the algorithm is restricted to block sizes that are a power of two.

Algorithm 1 outlines the procedure. We recursively continue contraction, until the node weight of a contracted node is equal to the requested block size. All nodes in a contracted node on the last level form the block in the initial partitioning (line 14). Note that for $k = \frac{n}{2}$ with perfect balance this would give us an exact solution, if an optimal matching algorithm is used. This motivates the intuition, that matched nodes could also form a part of a larger block.

In many cases we do not have a perfect matching, so nodes which are not incident to the same edge must be matched in order to ensure that the graph size is halved. We match unmatched nodes in two stages. First, we apply the method similar to 2-*hop matching* [30], which matches two nodes that share a common neighbor (line 7 - 8). This can be achieved in $\mathcal{O}(|V| + |E|)$ by scanning each $N(v)$ for $v \in V$ and matching as many as possible unmatched nodes per neighborhood. If there are no more such nodes, the remaining nodes are matched arbitrarily (line 9 - 10). In the case that the current number of nodes is odd, the last remaining node remains unmatched (line 11 - 12). An example execution of the algorithm is illustrated in Figure 4.

Since exact maximum matching algorithms have excessive running time, we use fast heuristics as explained in Section 3.2.2. We chose the Heavy Edge as a simple and fast algorithm and the Global Paths Algorithm as a more advanced technique for our evaluations.

Algorithm 1: Initial Partitioning - Matching Contraction

`Input`: Graph $G = (V, E), k = \frac{n}{2^l}$
`Output`: k-way partition $P = (V_1, V_2, \ldots, V_k)$

```
1  G' ← G
2  for i ← 0 to l do
3      M ← approximateMaxMatching(G')                    // heavy edge matching or gpa
4      for v ∈ V do
5          if v is not matched then                      // all neighbors of v are matched, since M is maximal
6              continue
7          N_1 ← {w | w ∈ N(v) and w is not matched}      // nodes that share v as neighbor
8          match elements of N_1 arbitrarily and add them to M
9      N_2 ← {v | v ∈ V and v is not matched}            // remaining nodes
10     match elements of N_2 arbitrarily and add them to M
11     if |V| is odd then                                // remaining node is alone in block
12         add unmatched node to M
13     G' ← contract(G', M)
14 P ← (G'.node(1), G'.node(2), ..., G'.node(k))
```

The runtime per level is dominated by the matching algorithm, since all other steps can be performed in linear time. There are only $\mathcal{O}(1)$ levels, since the block size is constant. In total

the runtime is $\mathcal{O}(|E|)$ or $\mathcal{O}(|E| \log |E|)$ depending on the matching algorithm.

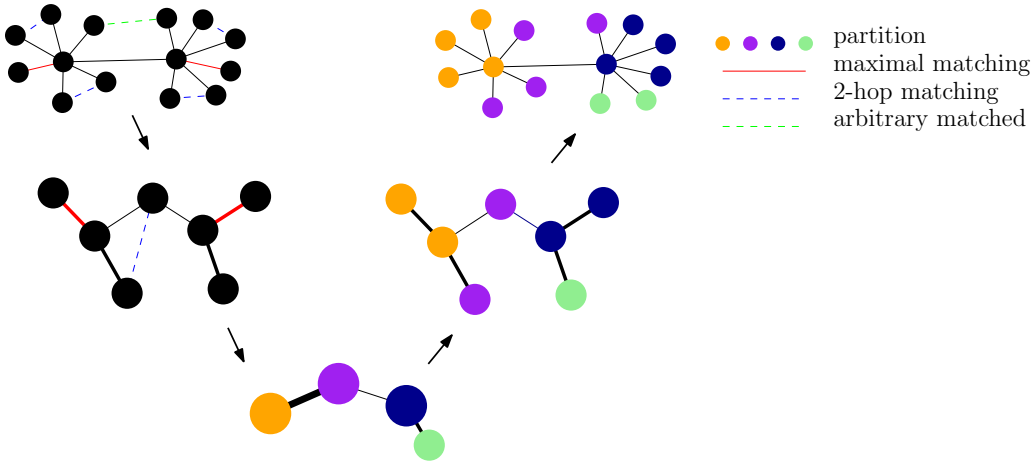


Figure 4: Example of Matching Contraction with two contractions.

4.2 Refinement

In the following various refinement algorithms are presented. Each describes one iteration, which is iterated until a maximal number of iterations is reached or the improvement over the last cut was smaller than a fraction α. Throughout this thesis we will refer to this procedure as *stopping rule*. All the techniques are local search algorithms using max gain moves to improve the current solution. To determine such a move for a node v of block b_v, it suffices to find a neighbor $w \in N(v)$ with block b_w, such that $\omega(v, V_{b_w})$ is maximized, since v is fixed in $g_{b_w}(v) := \omega(E(v, V_{b_w})) - \omega(E(v, V_{b_v}))$ and since neighbors of v contribute at least one edge that is removed from the cut, while non-neighbors do not contribute any edge. The edge cut $\omega(E(v, V_{b_w}))$ for $w \in N(v)$ can be easily calculated by iterating over $N(v)$ and storing the current cuts in a sparse hash map with the block ID as key, being very efficient for graphs with small max degree. Ties are broken randomly. As a consequence, more moves are considered and possible loops of zero gains moves are prevented. Similarly to Section 3.4.1 we call a move of a node v to a block V_i *eligible*, if $c(V_i)$ does not exceed L_{max} once v is moved to V_i.

4.2.1 Size Constrained Label Propagation

We will use the Size Constrained Label Propagation refinement algorithm mentioned in Section 3.4.1. Instead of maximizing $|N(v) \cap V_i|$, we maximize $\omega(v, V_i)$ the edge cut to cluster V_i. As described in Section 4.2, this is precisely the max gain block. Note that for unweighted graphs both variants are equivalent. Additionally, moves are only applied, if they improve the cut, i.e. gain ≥ 0, and if they are eligible. Zero gain moves are explicitly allowed, as we observed in experiments that it results in better cuts than only performing strictly positive gain moves. The intuition behind zero gain moves is, that they continuously change the partition of the graph and possibly make space for other moves. A pseudocode description is given in Algorithm 2.

In the original algorithm the order of node traversal was random. To determine the influence of the node ordering, we tried four different ordering strategies. *Normal* traverse the nodes by their node ID, resulting in better cache usage, since the nodes are stored continuous in an adjacency-array. *Random* randomly shuffles the nodes. *Chunk-Random* combines cache efficiency of *Normal* with the random shuffling of *Random*. The sequence of node IDs $1, 2, \ldots, n$

is cut into C continues equally sized intervals (chunks). Then each single chunk and the order of chunks is shuffled separately. Finally, there is the heuristic of traversing the nodes increasingly by their degrees [33]. We sorted the nodes by their most significant bit, since it is faster for large node-degrees. So nodes with degree $2^i \leq d < 2^{i+1}$ have equal key. E.g. $\{4, 5, 6, 7\}$ have all the same key in the sorting. In order to improve cache efficiency, additionally the node IDs are reordered by this sorting and the graph data structure is rebuild. In the following this variant is called *Pseudo Degree Sorting*.

Algorithm 2: Refinement - Label Propagation

`Input`: k-way partition $P = (V_1, V_2, \ldots, V_k)$
`Output`: k-way partition $P' = (V'_1, V'_2, \ldots, V'_k)$

```
1 order ← reorderNodes(V)        // normal, random, chunk random or pseudo degree sorting
2 P' ← P
3 for v ∈ order do
4 |   (found, gain, b) ← eligibleMaxGainMove(P', v)
5 |   if found and gain ≥ 0 then
6 |   |   move v to V_b
```

Parallelization. In each round, we iterate in parallel over all nodes. If a node is visited, analogous to the sequential case, the max gain move is computed. Then, we try to apply the move to the graph by first updating the block weight using an atomic compare-and-swap operation and then setting the new block for the node. This ensures that the balance constraint is not violated. If the move could not be applied, we calculate a new move and try again until either a move was successfully applied or no move was found. Note that moves with negative gain could be applied. This can happen if a node in the neighborhood of our current node changes its block after we computed the move, but before we could apply it. However, this rarely happens in practice and has not a big impact.

4.2.2 Mutation Refiner

At some point Label Propagation has run into a local minimum and can not find large improvements anymore. The Mutate Refiner combines the ideas of variable neighborhood search [34] with Label Propagation. In each iteration it mutates a small number of random nodes s and then performs Label Propagation in the region around the mutated nodes, possibly escaping local minima. Algorithm 3 outlines the procedure. For the mutation we apply a max gain move even if it has negative gain. (line 4 - 8). Up to $t \cdot |V|$, $t \in (0, 1]$ nodes are gathered by BFS for the region around the mutation nodes (line 10). The fraction t is useful to control the degree of locality. If no improvement was achieved by the mutation and Label Propagation, the mutation and the applied moves are reverted (line 16).

In order to compare different configurations of Mutation Refiner, each should move roughly the same amount of nodes in one iteration. For this reason we repeat the procedure of mutation and refinement until atleast $|V|$ not necessarily different nodes were used for Label Propagation (line 3, 17).

Algorithm 3: Refinement - Mutation Refiner

```
Input: k-way partition P = (V_1, V_2, ..., V_k), s, t, I
Output: k-way partition P' = (V'_1, V'_2, ..., V'_k)
1  P' ← P
2  j ← 0
3  while j < |V| do
4      M ← select s random nodes
5      for v ∈ M do                                          // perform mutation
6          (found, _, b) ← eligibleMaxGainMove(P', v)
7          if found then
8              move v to V_b
9                                                            // can be negative
10     R ← bfs(queue ← M, size ← t · V)                      // find refinement candidates by BFS
11     for i ← 0 to I do                                     // perform label propagtion on refinement nodes
12         for v ∈ R do
13             (found, gain, b) ← eligibleMaxGainMove(P', v)
14             if found and gain ≥ 0 then
15                 move v to V_b
16     revert this rounds moves, if total gain < 0
17     j ← j + |R|
```

4.2.3 Backward Path Refiner

The *Backward Path Refiner* (BPR) generalizes Label Propagation in two aspects. First, it considers sequences of moves that lie on a path formed by adjacent blocks, rather than movement of a single node. Thus, BPR overcomes the issues with a tight balance constraint that can prohibit single none-eligible moves. Moreover, one can allow negative moves in the sequence and apply the best prefix of moves similar to FM (Section 3.4.2). By not restricting negative moves and none-eligible, BPR can find more complex move sequences than Label Propagation. Secondly, BPR examines a set of moves between blocks and selects the move with maximal gain of this set.

Algorithm 4 outlines the method. A sequence of moves is constructed as follows. The algorithm starts from an underloaded block b and considers the set of all moves that move a node of $N(V_b)$ into V_b. It selects the move that has maximal gain, with ties broken randomly (lines 7 - 9). Then, this procedure is repeated recursively in the block, in which the moved node was part of (line 10), until a maximal *path length* γ is reached.

The algorithm is based on the idea that the application of one move enables new moves in near blocks, which is why the search continues in the former block of the moved node. In order to maintain the balance constraint, the current block must be underloaded, which is ensured by starting from an underloaded block and by the fact that the next block always is underloaded, since it moved a node out of its block. Note that this implementation assumes that when a node is moved to an underloaded block, the block will not become overloaded. This holds for unweighted instances, whereas for instances with node weight a move from $N(V_b)$ to a block V_b may not be eligible.

In one iteration, we traverse the blocks by their block IDs and start a path from the current block, if it is underloaded (line 4). This way the graph is refined equally in all places. Later we

will show the usage of negative moves, by testing the quality with and without negative moves. The tuning parameter γ controls the length of the path that is constructed. Greater γ results in more applied moves, but also in an increase in runtime.

Algorithm 4: Refinement - Backward Path Refiner

`Input`: k-way partition $P = (V_1, V_2, \ldots, V_k), \gamma$
`Output`: k-way partition $P' = (V'_1, V'_2, \ldots, V'_k)$

```
1  P' ← P
2  gains ← [0]
3  for j ← 1 to k + 1 do
4      if c(V_j) < L_max then
5          m ← j
6          for i ← 0 to γ do
7              w ← random element of {v ∈ N(V_m) | g_m(v) = max_{w∈N(V_m)} g_m(w)}
8              gains.pushBack(g_m(w))
9              move w to block m
10             m ← former block of w
11         u ← arg max_{p<|gains|} Σ_{l=0}^{p} gains[l]        // maximize prefixsum of gains
12         revert last(|gains| − u) moves
```

4.2.4 Forward Path Refiner

The *Forward Path Refiner* (FPR) works similar to the BPR but with two changes regarding the selection of moves and the handling of the best prefix of moves.

First, we explain how moves are selected. FPR searches for moves in the opposite direction as the BPR. Starting at a block b it computes the max gain move for every node of V_b and applies the move with maximal gain regardless of the balance constraint and possible negative gain of the move with ties broken randomly. The refinement is continued recursively in the block that received the moved node until a maximal path length γ is reached. Oppose to BPR, the number of moves considered at once is guaranteed to be constant due to our assumption that the block size is in $O(1)$. Note that the balance constraint in the current block may be violated. But since the next node is moved out of this block, only the last block of the current path may violate the balance constraint. Oppose to BPR, this implementation assumes that moving a node out of an overloaded block (created by FPR) restores the balance constraint. Again, this holds for unweighted instances, but can be a problem for instances with weighted nodes.

Secondly, like the BPR, we want to apply the best prefix of moves, guaranteeing that only move sequence with overall positive gain are applied. The problem is that the last block of a prefix of the path can be imbalanced. There are two variants to handle this. The naive variant is to apply the best prefix, that yields no imbalanced block. However, this method often reverts good move sequence, that are only imbalanced by one node. Given a move sequence, we call a move *balancing*, if the application of the move sequence and the *balancing* move has positive gain and yields a balanced partition. The idea of the second variant is to apply the best prefix, as in the naive variant, but instead of reverting the remaining moves, we try to find one balancing move, in order to "fix" a prefix of the remaining move sequence. To find a balancing move, we again consider every max gain move of every node of the current block, but exclude moves that are not balancing. If we found a balancing move, we apply it, otherwise we revert the last

move of the move sequence. If in the last step a move was reverted, we continue this procedure recursively with the remaining move sequence until either, we found a balancing move, or the move sequence is reverted completely. We call this variant FPR with *post-refine*.

In one iteration of refinement we start the FPR once from every block, to refine the graph equally in all places. In contrast to BPR, the start block must not be underloaded. The pseudocode is similar to Algorithm 4, with the two described changes.

Parallelization. We execute multiple runs of the FPR in parallel. To ensure that at most one thread works at a block, a thread must acquire the respective lock by an atomic compare-and-swap operation. In order to select a start block, a thread repeatedly tries to obtain a random block until it acquired the lock of some block. This way different runs operate on average distant from each other. A thread continues its path by computing a move like in the sequential algorithm and then tries to acquire the lock of the target block. During the computation of the best move of the block, locked blocks are ignored. If the lock of the target block was not acquired, the thread computes a new move in the same manner and tries again until it successfully applied a move. Locks are freed as follows: In the case that the maximal path length is reached or a balancing move is applied, all locks of modified blocks are freed. If the last move of a path is reverted, only the lock of the last block is freed. In order to decrease the number of locked blocks, we apply a sequence of moves directly, if it has overall positive gain and does not violate the balance constraint, rather than waiting until a maximal path length is reached. When this happens, all locks of the modified blocks except the last block are freed, since a path may continue from the last block. The earlier application of the moves does not reset to current path length. This parallelization assumes that there is a large number of blocks, such that the threads interfere rarely.

5 Experimental Results

5.1 Setup and Methodology

The proposed algorithms are implemented in C++ and compiled using g++-10.2 with flags `-O3 -march=native` turned on. We used Intel TBB [1] as parallelization library. The source code of our preliminary version is available at https://github.com/HaagManuel/kamipar/tree/master/kaminpar/super_large_k.

Setup. The experiments are performed on two different machines. Machine `A` has an Intel(R) Xeon(R) CPU E5-2670 v3 processor clocked at 2.3 GHz and 125 GB main memory. The second machine `B`, uses an AMD EPYC 7702 64-Core processor clocked at 2.592 GHz, 1044 GB main memory and 1 socket with 64 cores. Machine `A` will be used for parameter tuning experiments and machine `B` will be used for scalability experiments and the comparison to KaMinPar.

Instances. To evaluate our algorithms, we use the benchmark set used to evaluate the graph partitioner KaMinPar [19], referred to as set `A`, as well as two subsets of it. Set `A` is composed of 197 graphs including 129 graphs from the 10th DIMACS Implementation Challenge [7], 25 randomly generated graphs [16, 29], 25 large social networks [2, 31], and 18 graphs from various application domains [10, 43, 44]. For parameter tuning experiments we use a subset of 53 graphs with $|E| \in [10^6, 10^7)$, which we refer to as set `B`. Scalability experiments are performed on a subset of 38 graphs of set `A` with $|E| \in [10^7, 10^8)$ called set `C`. The subsets are chosen, such that

the largest graph in terms of edges of B and C take less than 10 minute respectively 1 hour with the sequential default configuration of our algorithm with BFS for initial partitioning and FPR for refinement. Due to limited availability of machine B, we had to choose a suitable subset such that our scalability experiments finishes within one week. Therefore, we had to exclude some instances that took more than 1 hour with one thread.

Methodology. Usually, graph partitioners are configured with the number of blocks k and the relative allowed imbalance ε. For a very large number of blocks it is more meaningful to set the upper bound for each block as $L_{\max} = N + b$, where N is the number of nodes allowed per block and b is the absolute allowed imbalance. The number of blocks is expressed by $k = \frac{c(V)}{N}$ and the relative imbalance ε is expressed by $\varepsilon = \frac{b}{N}$. A combination of a graph with, N nodes per block and absolute imbalance of b is called an *instance*. For each instance, we perform several runs with different random seeds and aggregate running times and edge cuts using the arithmetic mean over all seeds. To further aggregate over multiple instances, we use the harmonic mean for relative speedups, and the geometric mean for absolute running times and edge cuts.

Performance Profiles. We use *performance profiles* [11] to compare the solution quality of different algorithms. Let $\mathcal{A}$ be the set of algorithms we want to compare, $\mathcal{I}$ the set of instances and $q_A(I)$ the quality of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. Each algorithm is represented by a curve with τ on the x-axis and the fraction $\frac{|\mathcal{I}_A(\tau)|}{|\mathcal{I}|}$ on the y-axis, where $\mathcal{I}_A(\tau) := \{I \in \mathcal{I} \mid q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)\}$. Achieving higher fraction on lower values of τ is considered better. Note for $\tau = 1$, the y-value indicates the percentage of instances for which the algorithm performed best.

5.2 Parameter Tuning

For parameter tuning we partition the 53 graphs of set B in 5 runs with different seeds for each algorithm. Each graph is partitioned for $N \in \{16, 32, 64\}$, $b = 1$. The values of N are powers of two, since this is necessary to run the Matching Contraction algorithm. However, we leave the general case open for future work. Each refinement algorithm runs until the improvement in one iteration is less than $\alpha = 0.1\%$ of the previous edge cut or until a maximal number of iterations of 100 is reached.

The deviation from the balance constraint b is not set too high, since otherwise the different values of N would have high variety in relative allowed balance. However, b must be at least one to guarantee the existence of solution. The value of the minimum relative improvement threshold α is a trade-off between running time and solution quality and was set based on previous experiments.

5.2.1 Initial Partitioning

In the following we compare the initial partitioning algorithms described in Section 4.1. First, we determine the best configuration for BFS / greedy growing and the Matching Contraction algorithms separately. We will demonstrate that the edge cut after the initial partition is not always accurate on determining the quality of an initial partition algorithm. That's why we use performance profiles with the edge cut *after* refinement for comparison of solution quality. For the initial partitioning experiments, we use Label Propagation for refinement, since it is very fast. At the end of this section we compare the best initial partitioning algorithm to the Random partitioner.

BFS Configuration. We tested the two start node strategies `a` ∈ {`peripheral`, `random`} and the two next node strategies `b` ∈ {`border`, `random`} for the BFS partitioner. Figure 5 shows the quality of the algorithms after refinement, if we compare the values of `a` while fixing `b` = `border` (left) and if we compare the values of `b` while fixing `a` = `peripheral` (right). Clearly the start node strategy has no significant impact on the solution quality. Regarding the next node strategy, the `border` strategy produces on 95% of the instances the best solution, while the random strategy is on 50% of the instances within a factor of 2% of the best algorithm. The start node strategy does not influence the initial partitioning runtime, whereas the border next node strategy (0.12s geometric mean running time) is slightly faster than the random strategy (0.19s). For the upcoming experiments the default variant of BFS will be `a` = `peripheral` and `b` = `border`.

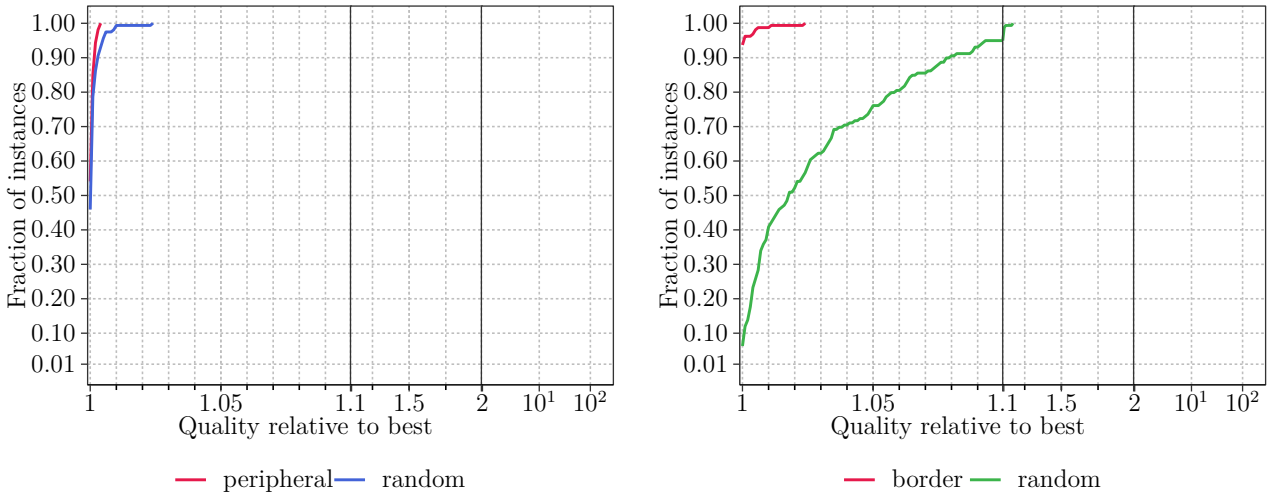


Figure 5: Quality of BFS partitioner with different start node strategies (left) and different next node strategies (right).

BFS vs. Greedy Growing Next we will compare the best BFS variant with the greedy growing algorithm, referred to as Greedy, that uses the same start and next node strategy. In Figure 6 we see that Greedy yields better initial cuts. It produces the best initial cut on 90% of the instances. However, surprisingly after refinement the big gaps in quality mitigate and BFS is even slightly better. To confirm the observation, we made the same experiment with BPR and FPR as refinement algorithm and got similar results (see Appendix A). The BFS partitioner produces on 80% of the instances solutions that are only 1% worse than the best solution, while the greedy partitioner achieves this result only for 55% of the instances. So the initial cut is not necessary an indicator for overall quality after applying refinement algorithms. In this case the BFS algorithm, that grows each block equally in all direction, provided better initial partitions for refinement than Greedy. One possible explanation for the significant increase in quality of BFS after refinement is, that Label Propagation considers the same nodes in the neighborhood of a node for a move as Greedy when building a block. So the refinement algorithm can construct similar blocks as Greedy. That Greedy after refinement is worse than BFS could be caused by decisions of Greedy that were locally good, but turned out to be bad overall.

Matching Contraction Configuration. In this experiment, we evaluate the influence of the matching algorithm on the Matching Contraction partitioner by comparing the simple heavy-edge matching, referred to as Heavy, with the the global path growing matching algorithm,

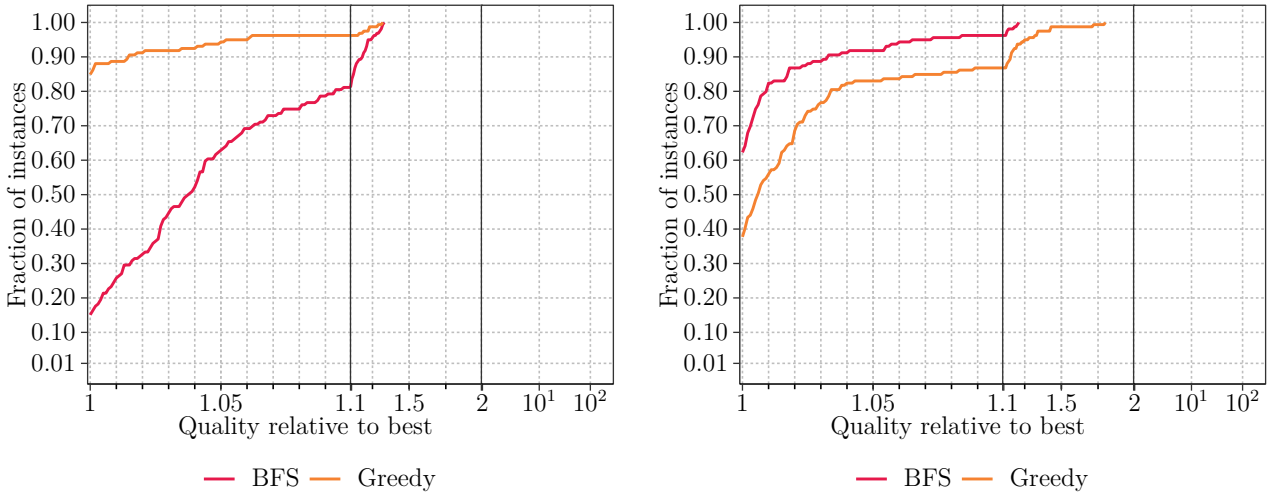


Figure 6: BFS vs. Greedy with initial partition cut (left) and cut after refinement (right).

referred to as GPA. Secondly, we test the impact of the 2-*hop-matching* [30], i.e. matching unmatched nodes within a distance of 2. We make use of the existing GPA implementation[1] [40], by calling a separate build with a system call. Additional time spend for the call was excluded in the runtime. The results are shown in Figure 7. In the left performance profile, we compare two variants using GPA as matching algorithm, but one uses additionally the 2-hop-matching. On 95% of the instances the variant using 2-hop-matching computes the best solution. The right plot shows the impact of the used matching algorithm, while both algorithms use 2-hop-matching. The variant using GPA computes the best solution on 80% of the instances. As we presumed 2-hop-matching and the use of a more advanced matching algorithm come with significantly amount of improvement of the solution quality. Looking at the average initial partitioning time, 2-hop-matching had no significant overhead, whereas the GPA comes with a cost in runtime on average of $5.08s$ or $5.12s$, with or without the 2-hop-matching respectively, where Heavy only takes $0.6s$ or $0.57s$.

When comparing GPA using 2-hop-matching with BFS, we observe a similar result as in the comparison of Greedy and BFS. The GPA variant like Greedy computes better initial cuts, but both have similar edge cuts after refinement (see Appendix A). Since BFS is considerably faster, we chose it as our default initial partitioning algorithm.

Quality of the Random Partitioner. Now we compare BFS with the Random partition. As shown in Figure 8 the random partitioner produces on 80% of the instances initial cuts that are more than twice as large as the best known solutions. After refinement 45% solutions of random are at least 10% worse than the best solution. The random approach is very fast on average (0.04s), but increases the refinement time by more than a factor of 2 compared to all other initial partitioning algorithms. This may be caused by the fact, that in the initial partition very few nodes have neighbors that are in the same block. So Label Propagation probably needs some time to reduce the noise of the random partition and to construct blocks that have some degree of locality.

5.2.2 Refinement

In this section, we will evaluate the refinement algorithms of Section 4.2. First, we test four different ordering strategies for Label Propagation. Secondly, we determine the influence of the

[1] https://github.com/schulzchristian/GPAMatching

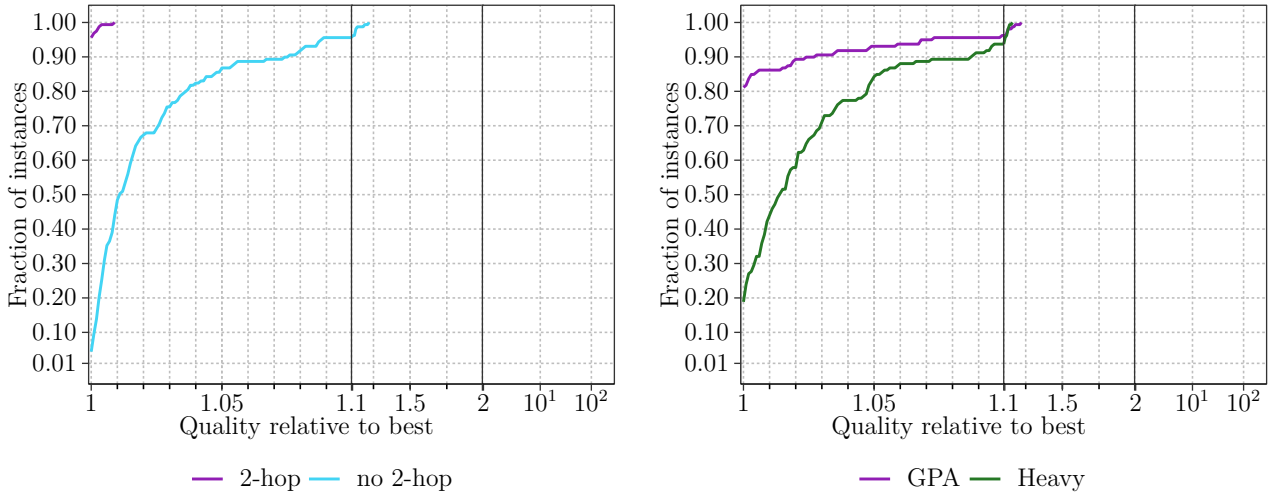


Figure 7: Comparison of Matching Contraction variants. 2-hop-matching (left), simple vs. advanced matching algorithm (right).

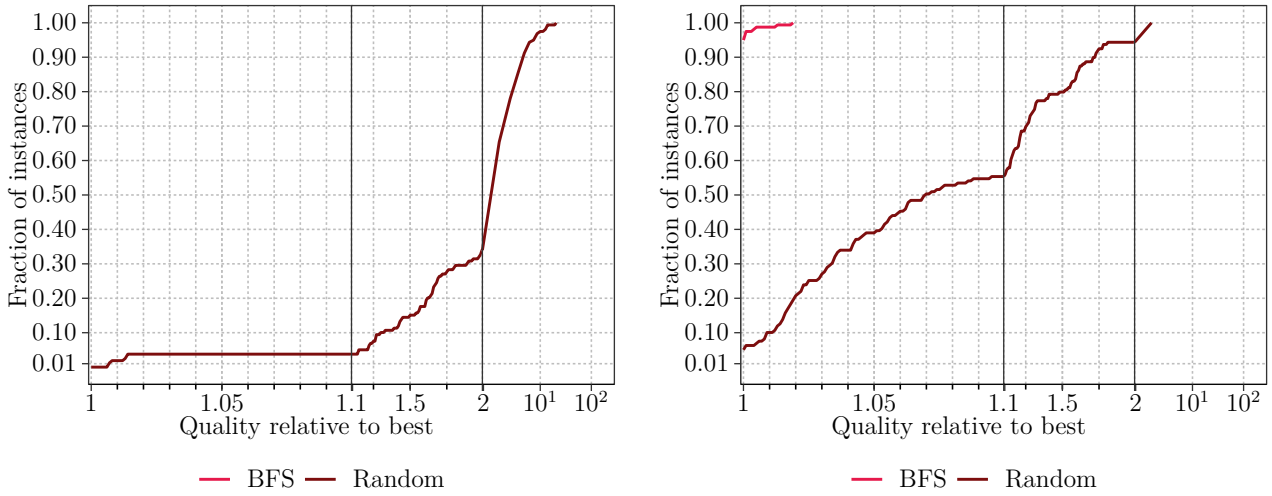


Figure 8: Quality of BFS and Random partitioner, initial partition cut (left), cut after refinement (right).

number of mutation nodes s and the fraction of refinement candidates t for Mutation Refiner. Then, we compare two variants of Backward Path Refiner to the Forward Path Refiner. And finally, we evaluate the trade-off between quality and running time of the minimum relative threshold α for Label Propagation and Forward Path Refiner.

Size Constrained Label Propagation. To determine the influence of the order in which Label Propagation (referred to as LP) iterates over the node, we tested the four ordering variants described in Section 4.2.1, namely input order (*Normal*), random shuffling (*Random*), shuffling equally sized chunks of nodes and the order of chunks (*Chunk-Random*) and sorting the nodes ascending by their most significant bit of their degree *Pseudo Degree Sorting*. For *Chunk-Random* we picked a chunk size of 1024. The input order of node IDs have some locality, and by shuffling only small chunks, later iterations over the node IDs might be more cache-friendly. Furthermore, we applied the reordering for *Random* and *Chunk-Random* only once at the start, to avoid the overhead in runtime due to shuffling. We made a run with random shuffling before each LP iteration (*Random every round*), to see how this changes affects the quality and the

running time. For *Pseudo Degree Sorting* we use the existing parallel implementation[2] that is used in KaMinPar to reorder the graph. Meyerhenke et al. [33], who used Label Propagation for coarsening, also tried different orderings and came to the conclusion that sorting by node degrees provided significant improvement, while other orderings like *Random* did not have an impact. However, in our experiment all ordering perform similarly Figure 9. The variants *Chunk-Random* and *Pseudo Degree Sorting* have about the same average refinement time of 1.71s and 1.7s respectively. Due to many cache misses, *Random* takes more than double the runtime oppose to the other variants with 3.82s. *Random every round* has the worst runtime of 4.58s and is 20% slower than *Random*. We picked the *Normal* variant as the default, as it has the fasted refinement time of 1.47s on average.

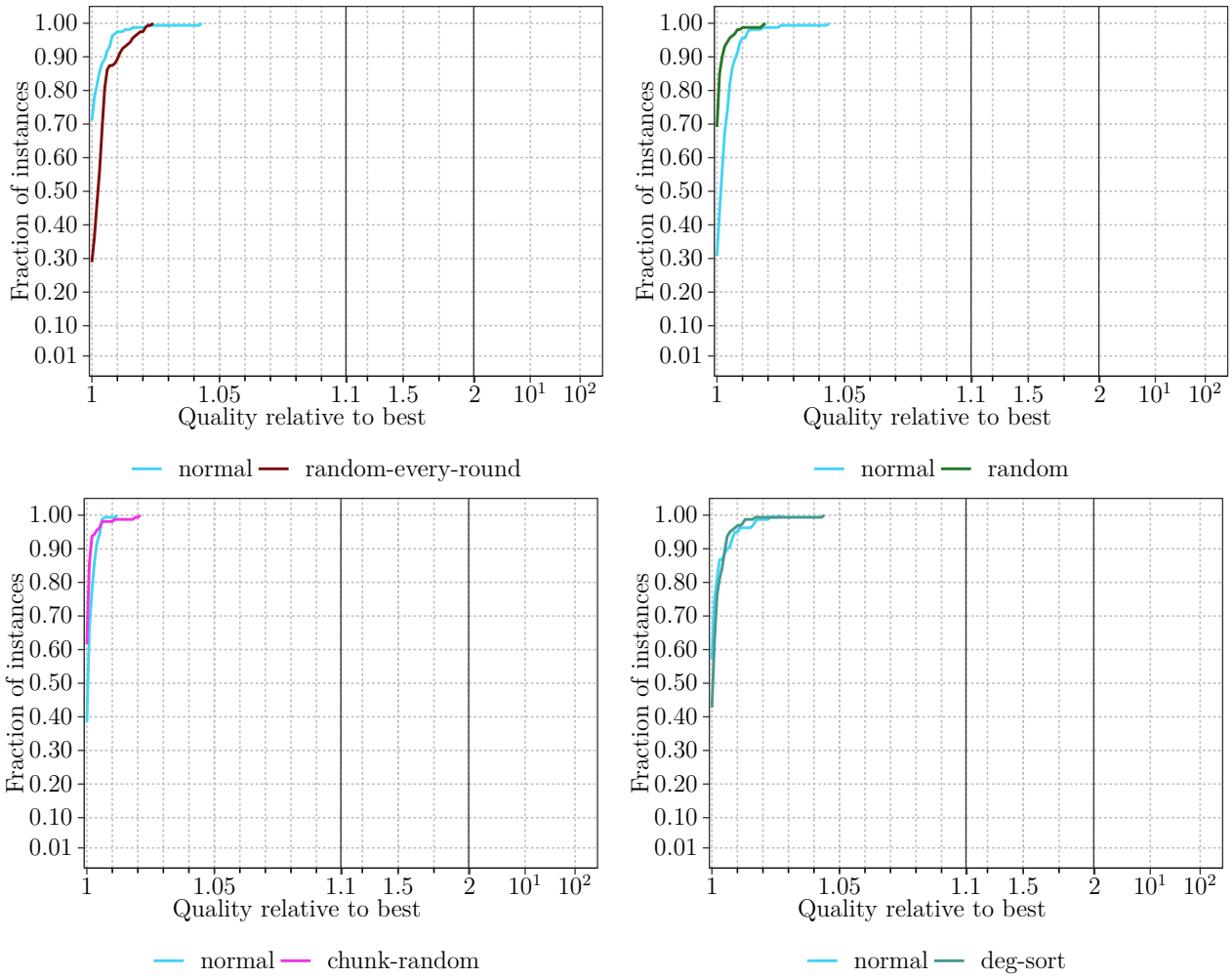


Figure 9: Quality of Label Propagation with different node orderings.

Mutation Refiner In this experiment we want to evaluate the impact of the number of mutation nodes s and the fraction of refinement candidates t used in the Mutation Refiner (referred to as MR). The number of iterations of Label Propagation is fixed at 8 and was set based on preliminary experiments. Recall, that for the comparability of the different values of t, the procedure of mutation and Label Propagation around the mutation nodes is iterated until at least $|V|$ not necessarily different nodes were used for Label Propagation. So for example one repetition of $s = 64$, $t = 1\%$ performs 64 mutations and then refines the region

[2] https://github.com/KaHIP/KaMinPar/tree/v1.0/kaminpar/algorithm/graph_utils.cc

around the mutation nodes. Since in each repetition the region for refinement has a size of 1% of the total graph, the procedure is repeated about 100 times. Note that this counts as one iteration for the stopping rule. First, we tested $t \in \{0.1\%, 1\%, 10\%\}$. As shown in Figure 10, $t = 0.1\%$ yields the best quality out of the three values, with 55% best solutions. The better quality comes at the cost of longer running times, which steadily increased with $17.46s$ $25.94s$, $35.52s$ for $t = 10\%, 1\%, 0.1\%$. The reason for this is that more iterations of the Mutation Refiner are executed, due to the stopping rule: On average 16.24, 21.03, 26.26 iterations for $t = 10\%, 1\%, 0.1\%$. Recall that the stopping rule starts another iterations, if the last improved the cut by at least α, which for this experiment is set to $\alpha = 0.1\%$. Similar behavior regarding quality and runtime can be observed if we fix $t = 0.1\%$ and test different values of the number of mutation nodes $s \in \{4, 16, 64\}$. The number of average iterations also increases: 19.17, 22.90, 26.26 for $s = 4, 16, 64$ and therefore the runtime. A higher number of mutations nodes s and a higher degree of local refinement controlled by t results in better quality. We decided not to investigate the parameters of Mutation Refiner in more detail, since the improvement over Label Propagation (referred to as LP) are moderate, as shown in Figure 11 and LP is more than 20 times faster.

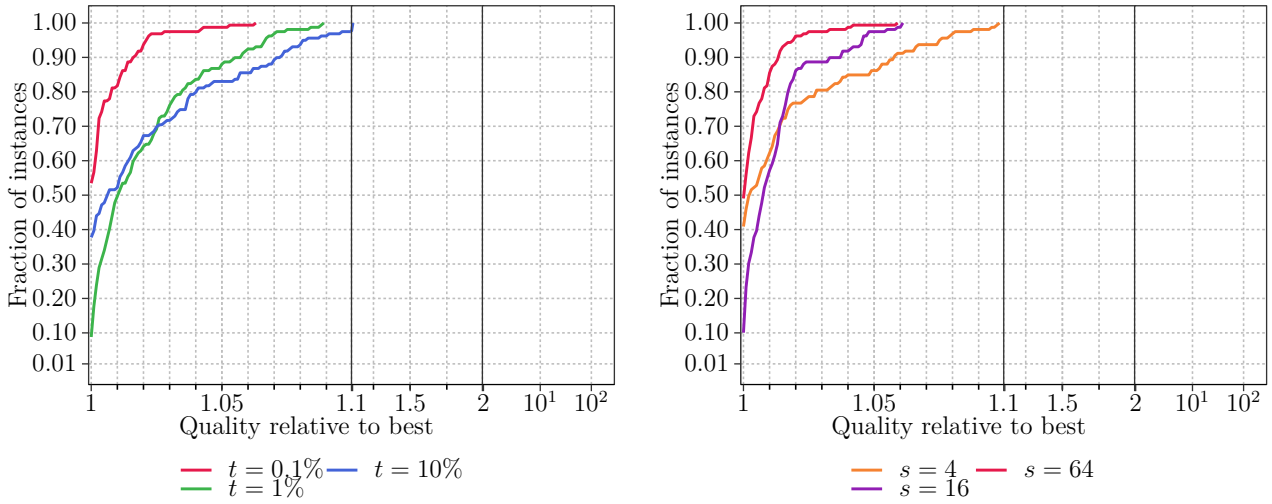


Figure 10: Mutation Refiner with different fraction of refinement candidates and $s = 64$ (left) and different number of mutation nodes and $t = 0.1\%$ (right).

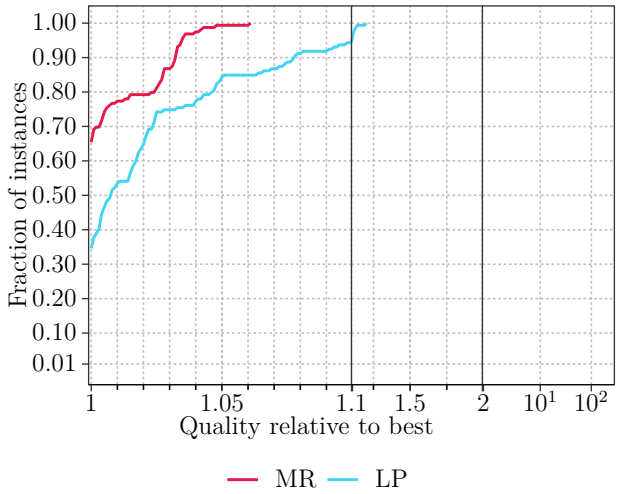


Figure 11: Comparison of Mutation Refiner with $s = 64$, $t = 0.1\%$ and Label Propagation.

Comparison of Path Refiner variants The goal of this experiment is to determine the best algorithm of the Path Refiner variants described in Sections 4.2.3 and 4.2.4 We tested three variants:

- BPR is the Backward Path Refiner algorithm from Section 4.2.3.
- BPR+ works like BPR with the only difference that moves with gain < 0 are not applied. If no positive gain move can be found, the search is stopped early.
- FPR is the Forward Path Refiner algorithm with *post-refine* described in Section 4.2.4.

Algorithm	Runtime $t[s]$			
	$\gamma = 4$	$\gamma = 8$	$\gamma = 16$	$\gamma = 32$
BPR+	7.70	10.57	15.04	**22.16**
BPR	9.42	14.15	**21.93**	35.24
FPR	11.31	**18.44**	31.38	55.21

Table 1: Average runtime of Path Refiner variants by path length γ. Running times of the configuration used in Figure 12 are marked bold and were selected, such that each path refiner runs for roughly the same amount of time.

As shown in Table 1, the algorithms differ significantly in running time for same values of the path length γ. FPR is consistently the slowest, followed by BPR and BPR+. In order to give each algorithm roughly the same amount of running time, we picked $\gamma = 8, 16, 32$ for FPR, BPR and BPR+ respectively (bold in Table 1). Figure 12 shows the results. Although FPR has the lowest average runtime with $18.44s$, it outperforms the other two variants. It computed 60% of the best solutions, whereas roughly 50% of the solutions of BPR and BPR+ respectively lie within a range of 1% of the best solution. All path refiners significantly outperform the Label Propagation algorithm. However, Label Propagation is an order of magnitude faster than the path refiner variants.

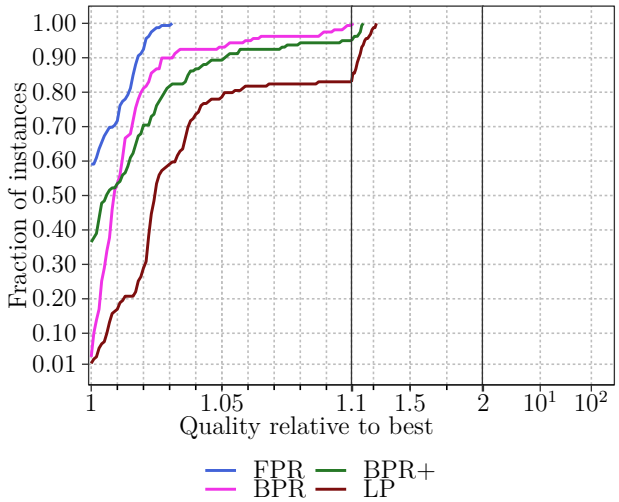


Figure 12: Comparison of Path Refiner variants and Label Propagation.

Figure 13 shows the quality of FPR with different γ relative to $\gamma = 8$. The improvement from 4 to 8 and 8 to 16 are very small. We set the default value of γ for FPR to 8, since the improvements in quality of larger γ are too small for the cost in runtime. An interesting observation is that the average number of performed refinement iterations decreases with increasing γ : $18.05, 15.68, 13.82, 12.25$, for $\gamma = \{4, 8, 16, 32\}$. So longer *path length* leads to more

improvement per round. But such variants run into similar optima as variants with shorter path length, as their solution quality does not differ much. In the following we will refer to Forward Path Refiner simply as Path Refiner (PR).

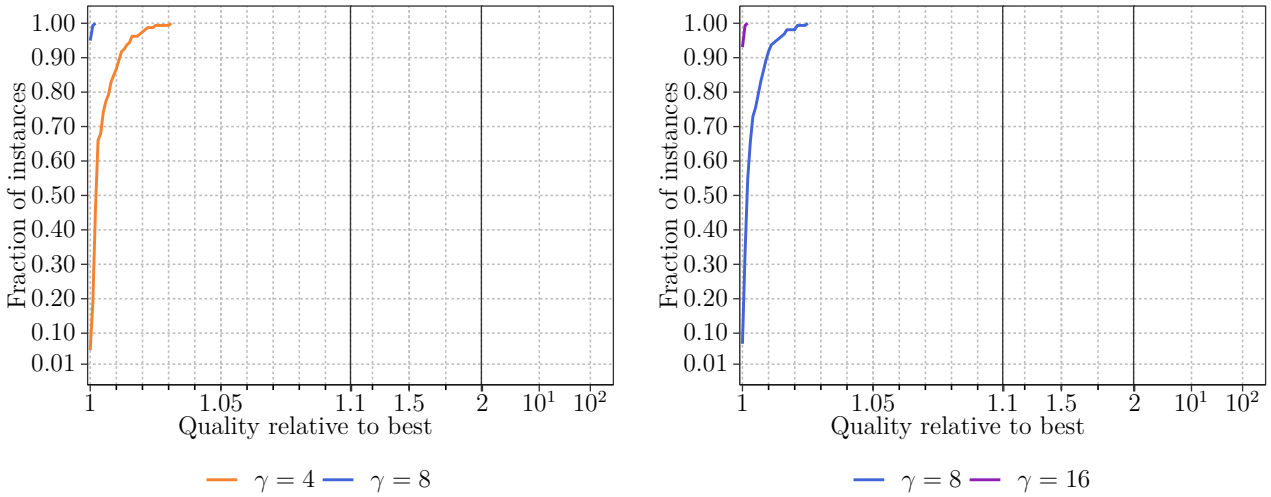


Figure 13: Quality of Forward Path Refiner with $\gamma = 4, 16$ relative to $\gamma = 8$.

Configuring minimum relative improvement threshold α. Now we will evaluate the trade-off between quality and running time of the minimum relative improvement threshold $\alpha \in \{1\%, 0.5\%, 0.1\%, 0.05\%, 0.01\%\}$ for Label Propagation (LP) and Path Refiner (PR). Table 2 shows the runtime of the two algorithms with different values of α. We observe that the runtime from $\alpha = 1\%$ to $\alpha = 0.1\%$ increases by a factor of ≈ 2.86 (LP), ≈ 3.45 (PR) and from $\alpha = 0.1\%$ to $\alpha = 0.01\%$ by a factor of ≈ 2.34 (LP), ≈ 3.31 (PR). The increase for PR are higher than for LP, since PR can escape local minima to a certain extent and thus can find improvement for a longer amount of time. While the average runtime of LP for larger $\alpha \in \{0.05\%, 0.01\%\}$ is still fast, PR becomes very slow.

Figure 14 (left) shows the quality of LP with different α. The algorithm with $\alpha = 0.01\%$ computes on all instances the best solution (thus no curve visible in the plot). $\alpha \in \{0.05\%, 0.1\%\}$ are comparable to the best configuration, finding on 60% of the instances solutions that are only 1% worse than the best. However, $\alpha \in \{0.5\%, 0.1\%\}$ produce considerably worse solutions. The plot for PR looks similar. There is a quality-trade-off between $\alpha = 0.1\%$ and $\alpha = 0.01\%$ for LP, with improvement in the order of 1% at the cost of roughly double the runtime. But for PR the runtime becomes too high to justify the small increase in quality. For larger values of α the loss in quality for LP and PR is too high to make up for the faster runtime. We prefer the faster running time for LP, so the default value is $\alpha = 0.1\%$ for PR and LP.

Algorithm	Runtime $t[s]$				
	$\alpha = 1\%$	$\alpha = 0.5\%$	$\alpha = 0.1\%$	$\alpha = 0.05\%$	$\alpha = 0.01\%$
LP	0.51	0.71	1.46	1.98	3.49
PR	5.22	7.46	18.03	26.62	59.72

Table 2: Average runtime of Label Propagation (LP) and Path Refiner (PR) by minimum relative improvement threshold α.

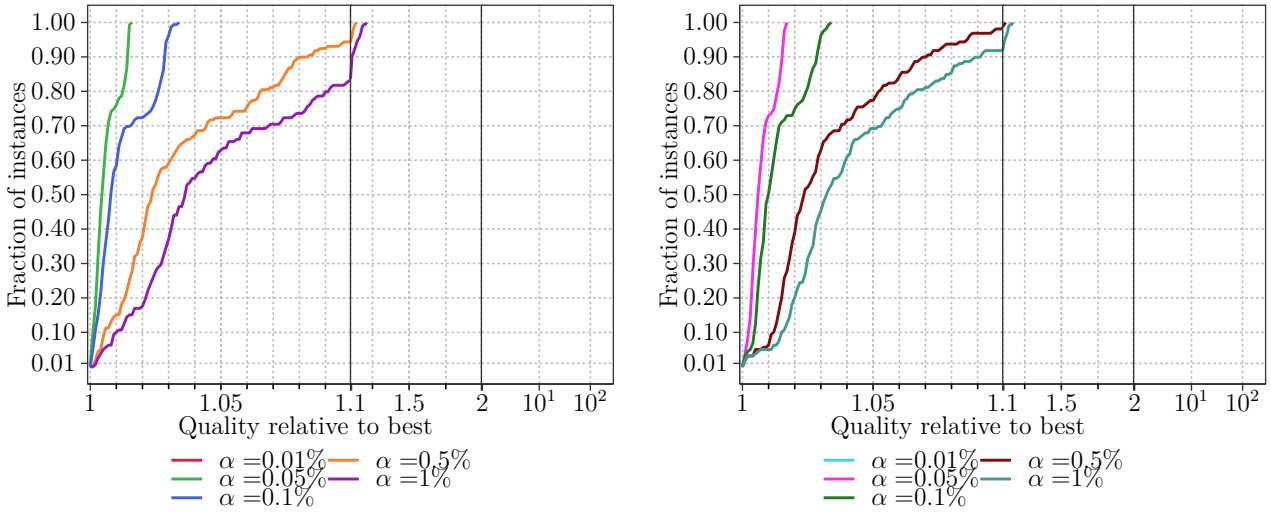


Figure 14: Label Propagation (left) and Path Refiner (right) with different thresholds α.

5.3 Scalability of the Parallel Implementation

Figure 15 shows the scalability of Label Propagation (LP) and Path Refiner (PR) with BFS initial partitioning for $N \in \{16, 32, 64\}$ and imbalance parameter $b = 1$ on set B. The experiments are performed with five repetitions per instance and $p \in \{1, 4, 16, 64\}$ threads. In the plot, we represent the speedup of each instance as a point and the cumulative harmonic mean speedup over all instances with a single-threaded running time $\geq x$ seconds with a line. The overall harmonic mean speedup without super-linear speedups with $p \in \{4, 16, 64\}$ threads of LP are $3.56, 11.58, 18.60$ and for PR $3.73, 11.74, 28.72$. Our algorithms do not perform expensive arithmetic operations. Hence, perfect speedups are not possible due to limited memory bandwidth.

Our BFS partitioner shows moderate speedups. However, its running time is an order of magnitudes faster than our refinement algorithm. Thus, it does not influence the overall scalability of our system. There are two possible reasons for this behavior. First, each thread needs a BFS queue and a queue for the border nodes. These queues potentially can use $\mathcal{O}(|V|)$ memory per thread, which can increase the number of cache-misses with increasing number of threads. Secondly, when many nodes are assigned, different BFS could interfere with each other by competing over nodes.

For PR we observe super-linear speedups on 14 instances. These instances are mainly complex networks characterized by highly-skewed vertex degree distributions. The cause for the super-linear speedup is, that threads interfere more with each other due to extensive block-level locking. Less improvement per iteration is found, so the stopping rule terminates the algorithm earlier than in the single-threaded setting.

The loss in quality with $p \in \{4, 16, 64\}$ threads for LP and $p \in \{4, 16\}$ threads for PR compared to single-threaded execution is negligible. PR with $p = 64$ threads produces moderately worse solutions than the single-threaded execution. Detailed performance profiles can be found in Appendix B.

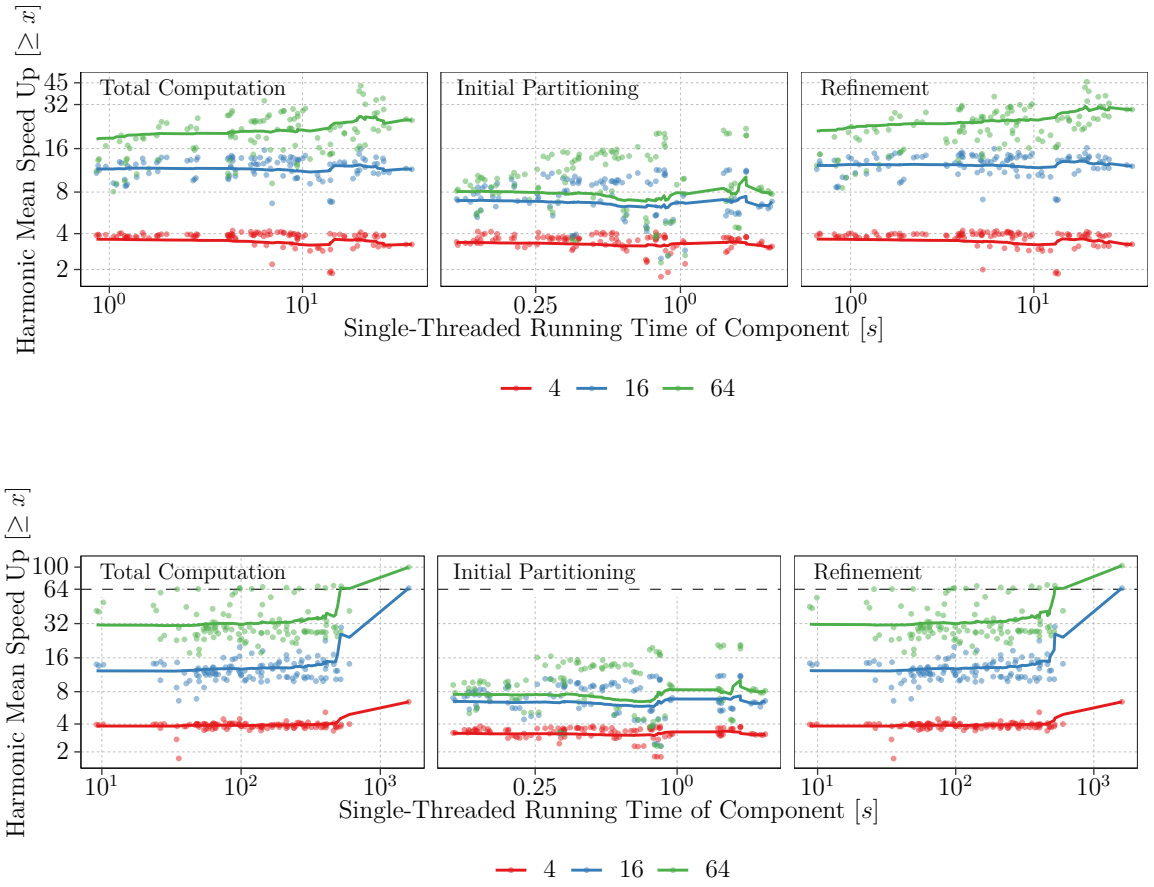


Figure 15: Self-relative speedups Label Propagation (top), Path Refiner (bottom) with BFS as initial partitioning

5.4 Comparison to KaMinPar

We compare two of our configurations to KaMinPar from Section 3.6, in order to evaluate if non-multilevel techniques are faster and potentially better than KaMinPar for very large k. The first configuration uses parallel Path Refinement, as it performs the best in terms of solution quality and our second configuration uses parallel Label Propagation, since the algorithm is fastest and offers a good trade-off between quality and running time. The experiments are executed with 64 threads.

As a benchmarkset, we use a subset of set `A` composed of 141 graph with at least 10^5 nodes. We excluded small graphs from set `A`, since in applications of parallel computing the graphs are usually large. Each graph is partitioned for block sizes $N \in \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ and $\varepsilon = \max(0.03, \frac{1}{N})$. We run each algorithm 5 times on each instance initialized with different random seeds and use average runtime and edge cut for our evaluations. Since KaMinPar can only configure imbalance by ε, we use ε instead of b to express imbalance. The imbalance ε is set this way, such that an imbalance of at least one node is allowed to guarantee existence of solutions and whenever possible, an imbalance up to 3% is allowed as in the experiments in [19].

As shown in Figure 16, our algorithms perform better than KaMinPar on very small block sizes $N \leq 4$. LP is comparable to KaMinPar for $8 \leq N \leq 16$ and PR for $16 \leq N \leq 32$. For block sizes greater than 64, KaMinPar significantly outperforms both non-multilevel algorithms.

A more detailed insight shows the performance profiles in Figure 19. Compared to LP, KaMinPar produces about $18\%, 30\%, 45\%, 60\%, 75\%$ of the best solutions for $N \in \{2, 4, 8, 16, 32\}$, whereas compared to PR, KaMinPar produces $25\%, 28\%, 32\%, 48\%, 65\%$ of the best solutions for the same values of N. For $N \geq 64$, KaMinPar computes more than 75% of the best solutions than LP or PR respectively. Further, for $N \geq 256$, 50% of the solutions by non-multilevel techniques are at least 10% worse than KaMinPar.

Figure 20 shows the geometric mean runtime of the three algorithms for the different block sizes. KaMinPar is getting faster with increasing block size. PR is faster than KaMinPar for block sizes smaller than $N = 64$. LP, due to it's simplicity, has constant geometric mean runtime of $0.3s$ over all tested block sizes. Looking at Table 3, we observe that LP and PR achieve increasingly faster running time relative to KaMinPar with decreasing block size. For the block sizes $N \leq 8$, PR is on average 4 times faster than KaMinPar, and for $16 \leq N \leq 32$, by a factor of 1.5. LP is for $N \leq 32$ an order of magnitude faster than KaMinPar. For $8 \leq N \leq 32$ the better quality of PR over LP is preferred, but LP can still be useful, if very fast running time is a requirement for the application.

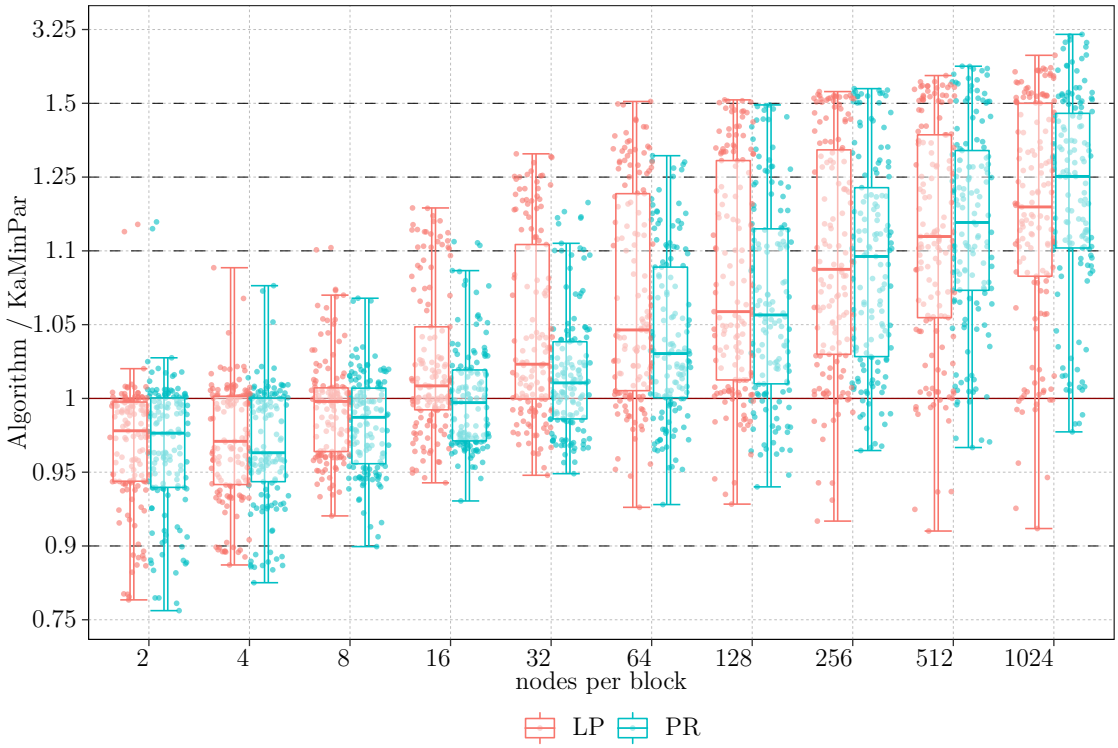
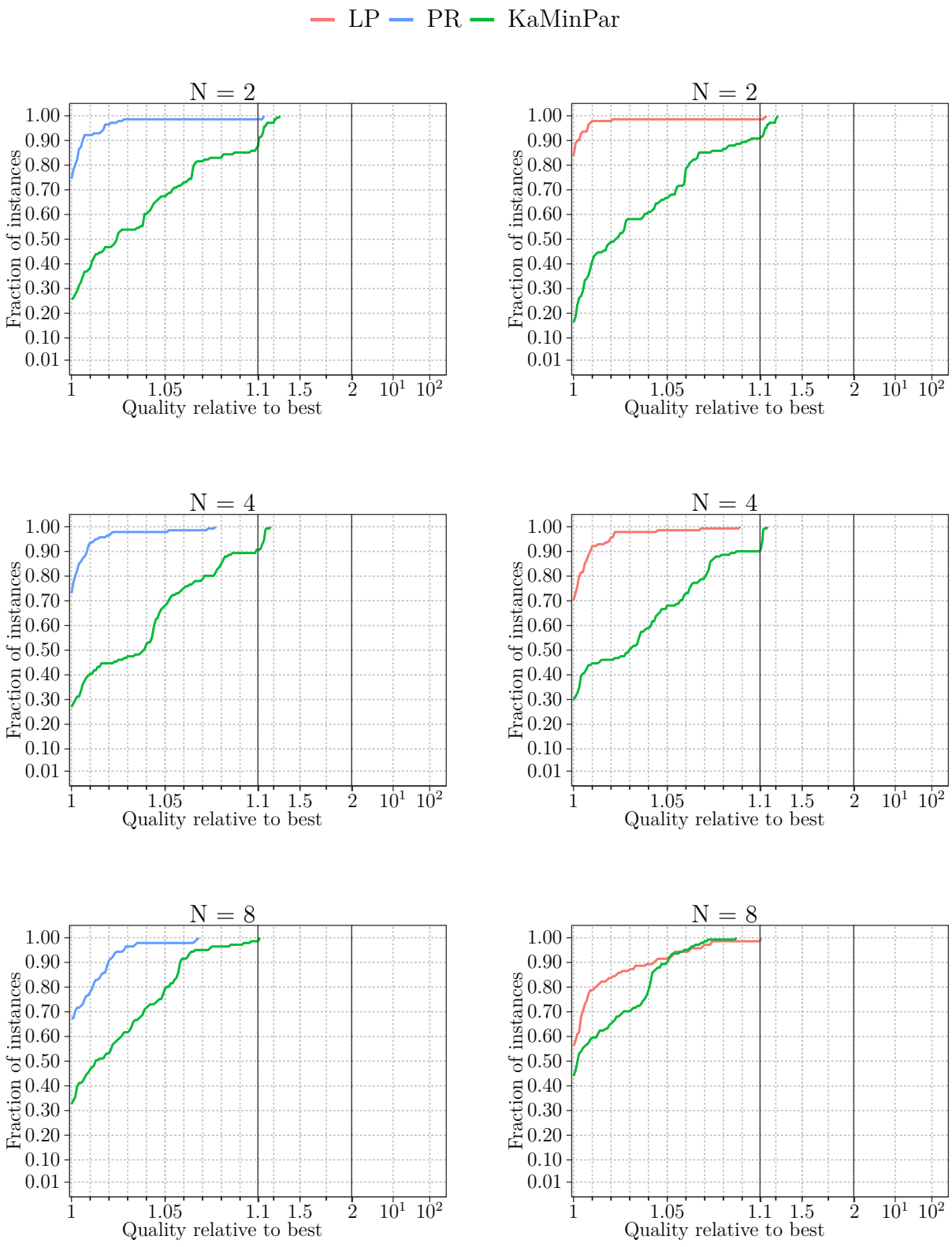
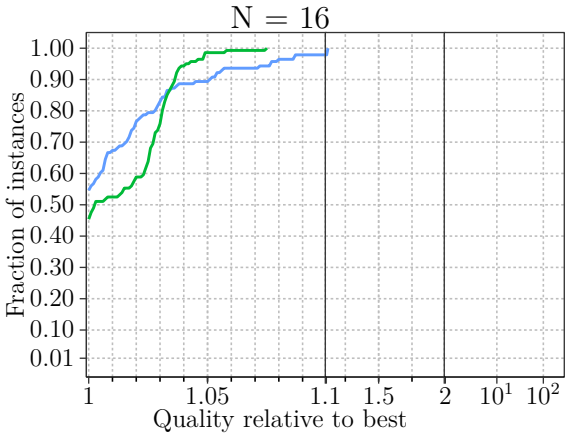
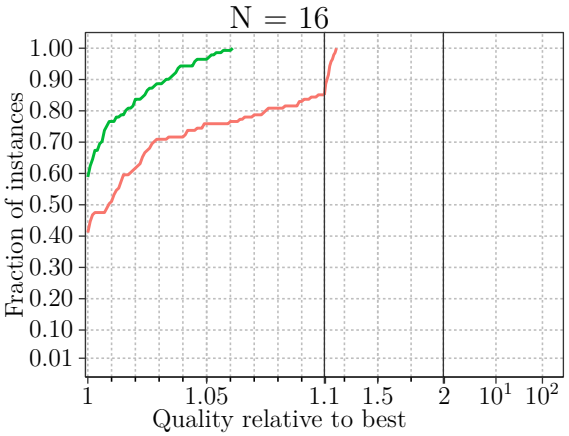
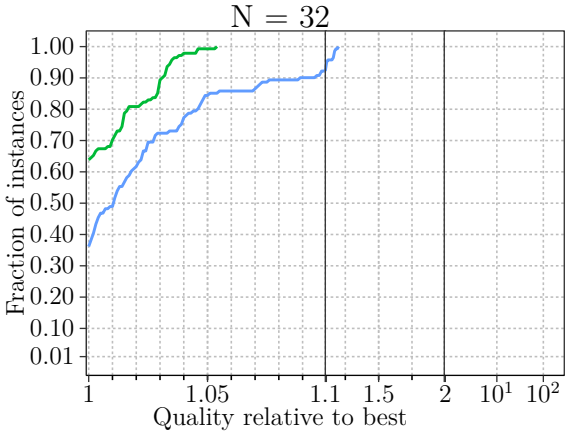


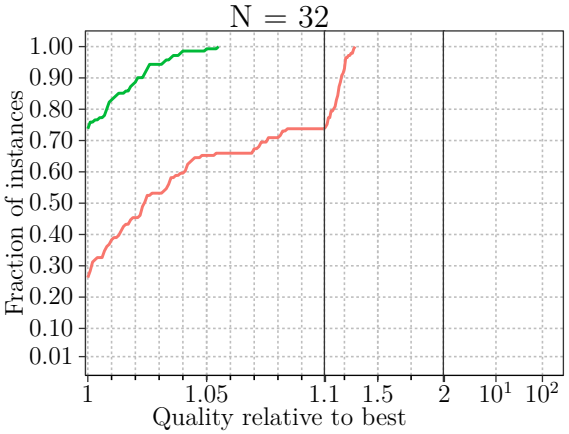
Figure 16: Ratio of cut between Label Propagation and KaMinPar and Path Refiner and KaMinPar per instance.

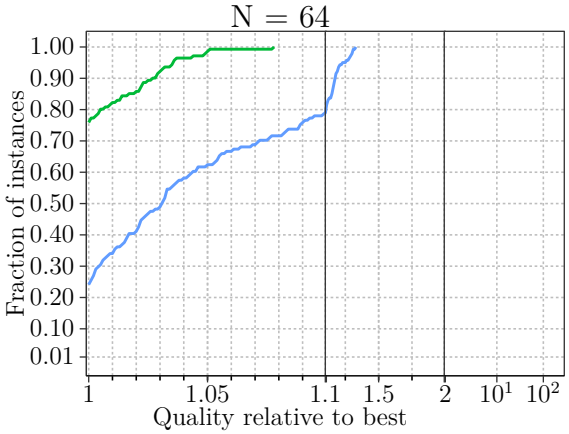

LP
PR
KaMinPar
N = 2
N = 2
N = 4
N = 4
N = 8
N = 8
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best

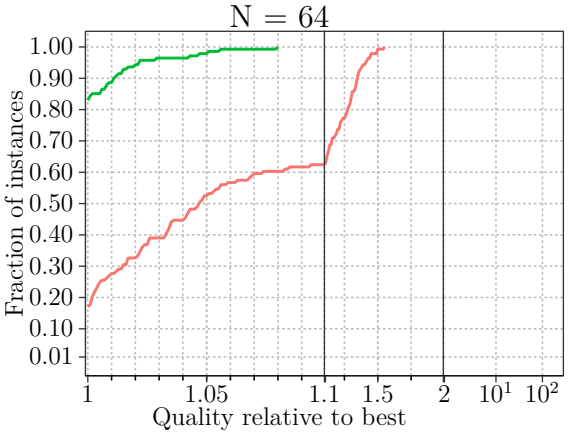

N = 16
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best

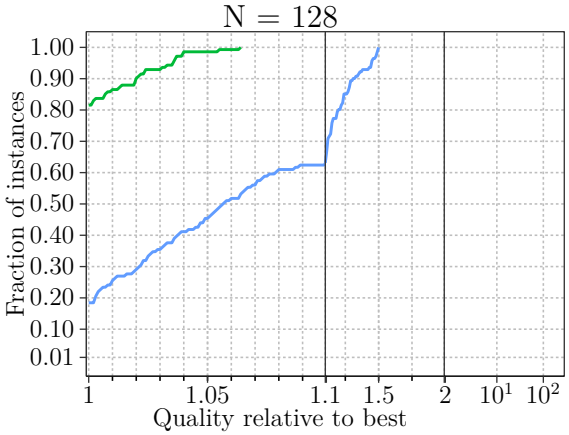

N = 16
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best

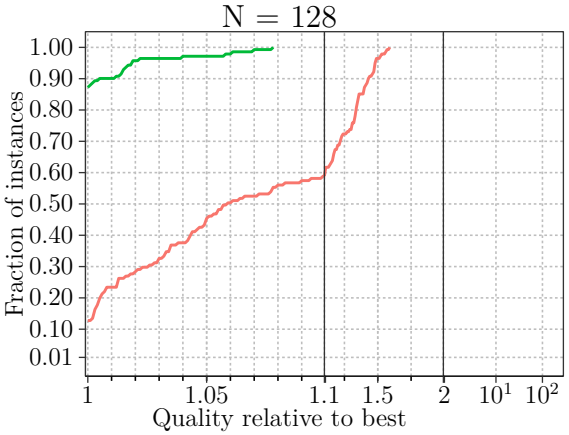

N = 32
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best


N = 32
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best


N = 64
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best


N = 64
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best


N = 128
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best


N = 128
Fraction of instances
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.01
1
1.05
1.1
1.5
2
10^1
10^2
Quality relative to best

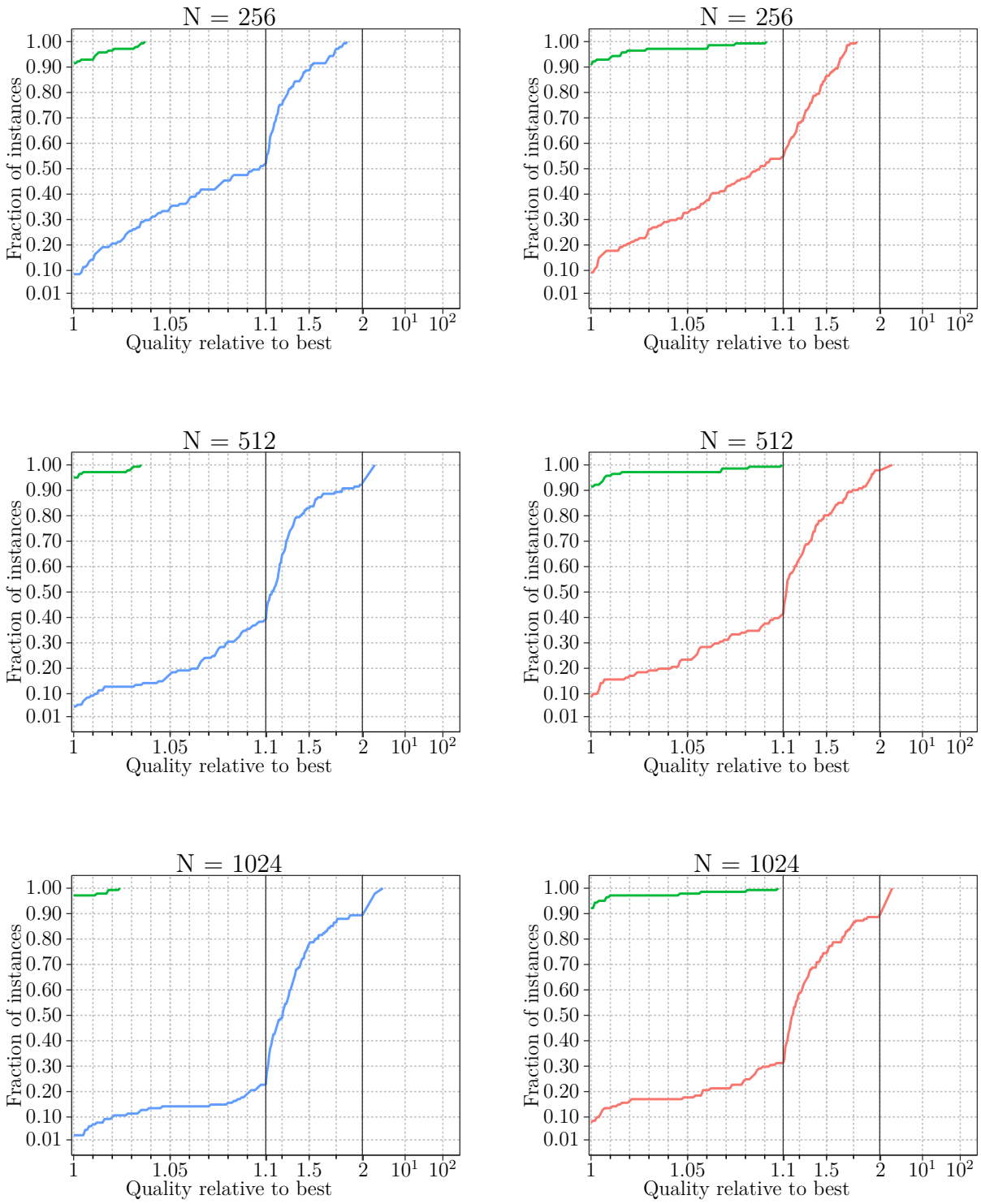


Figure 19: Performance profiles by node per block N comparing Label Propagation relative to KaMinPar (left), Path Refiner to KaMinPar (right).

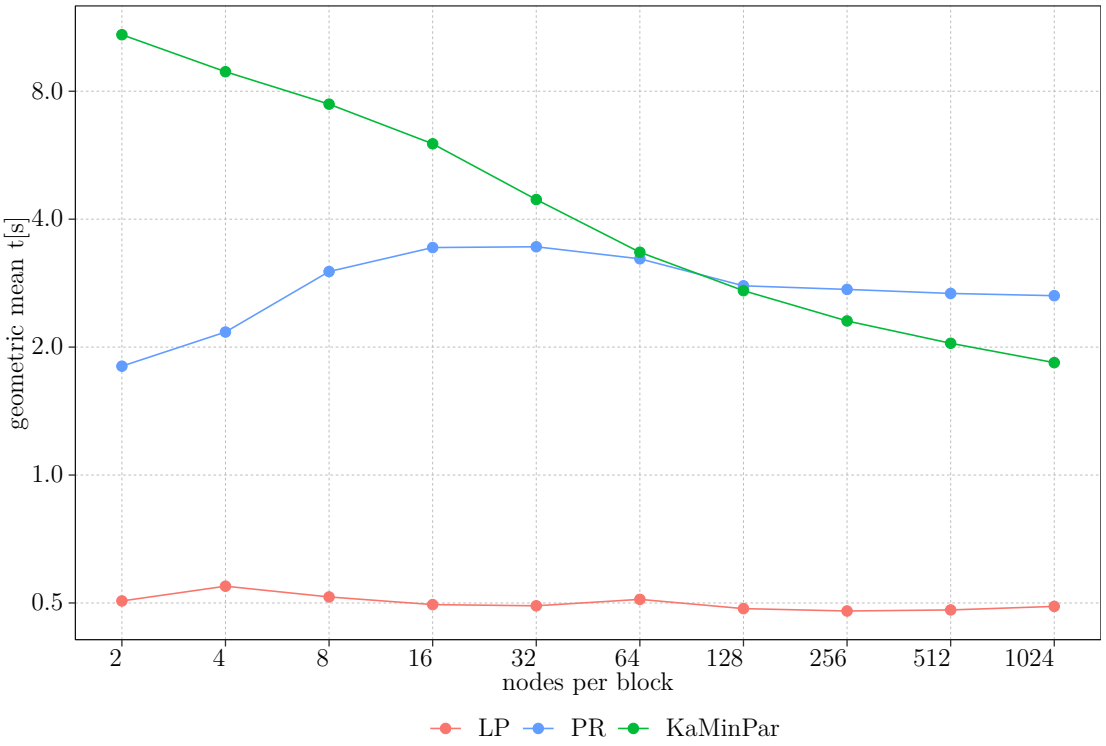


Figure 20: Geometric mean runtime of KaMinPar, Label Propagation and Path Refiner for different block sizes.

N	Runtime $t[s]$	Speedup	
	KaMinPar	LP	PR
2	10.87	36.05	6.02
4	8.89	28.65	4.10
8	7.46	24.68	2.48
16	6.02	19.64	1.75
32	4.45	14.53	1.29
64	3.34	10.77	1.04
128	2.72	9.15	0.97
256	2.30	7.81	0.84
512	2.04	6.88	0.76
1024	1.84	5.95	0.70

Table 3: Runtime of KaMinPar and speedup of Label Propagation and Path Refiner over KaMinPar for different block sizes N

6 Conclusion and Future Work

In this thesis, we presented and evaluated a shared-memory framework that can partition large graphs into a very large number of blocks, so each block contains only a small number of nodes. To this end, our algorithm partitions the graph directly into the desired number of blocks (*initial partitioning*) and afterwards, uses a novel local search algorithm to improve the quality of the partition (*refinement*). We showed through our experiments that our algorithm produces significantly better solutions than KaMinPar for very large k while being considerably faster.

For *initial partitioning*, we tested four different algorithms. A random partitioner, which randomly assigns nodes to blocks. Based on the graph growing techniques described in Section (3.3.1), we implemented a Breath First Search (BFS) partitioner, that grows each block by one separate BFS traversal. Further, we tested a greedy growing method that works similar to the BFS partitioner, but moves the node with the smallest increase in edge cut to the current block. Last, inspired by matching-based coarsening, we implemented the matching-based initial partitioner. The approach is restricted to the case where the number of nodes in each block is a power of two (general case left open for future work) and compute and contract a perfect matching until the number of nodes of the graph is equal with our desired number of blocks. Since perfect matchings rarely exists, we first compute a maximal matching and extend it to a perfect matching by matching the remaining nodes despite not being adjacent to each other.

For *refinement*, we tested four algorithms and repeated them until a maximal number of iterations is reached or the relative improvement found was less than a certain threshold. The first algorithm is the well-known size-constraint label propagation algorithm (LP) [33]. In one iteration the nodes are traversed in some order and a node v is moved to the block V_i minimizing the edges in the cut, with ties are broken randomly. The second algorithm called Mutation Refiner combines the ideas of Label Propagation and variable neighborhod search. In each iteration some nodes are mutated, by moving them to an adjacent block which can worsen the solution quality, and then Label Propagation is performed in a region around the mutated nodes, possibly escaping local minima. The last two algorithms are novel algorithms which we refer to as path refiner. They construct a sequence of moves that form a path on adjacent blocks until a maximal length is reached. We use the idea of traditional FM local search to allow intermediately worsen solution quality in the sequence and apply the best prefix of moves to achieve overall positive gain. Starting at an underloaded block V_i the Backward Path Refiner (BPR) moves the node $v \in N(V_i)$ with maximal gain $g_i(v)$ to V_i and continues the procedure recursively in the block in which v was part of. Whereas the Forward Path Refiner (FPR) starts from some block V_i, moves the node $v \in V_i$ with the maximal gain $g_j(v)$ to V_j and continues recursively with block V_j. The last block of the path constructed by FPR can be imbalanced, which is handled by finding a move from the last block to an underloaded block, if possible. Otherwise the last move is reverted until such a move was found or the complete sequence is reverted.

We also parallelized our most promising algorithms (BFS, LP and FPR) to handle large input graphs. BFS scales moderately for higher number of threads. But since the running time is an order magnitude faster than our refinement algorithms, this does not affect the overall scalability of our algorithms. Our parallel algorithm using FPR scales well with an overall harmonic speedup of 28.7 for 64 threads and our configuration using LP scales acceptable with an average speedup of 18.6. Finally, we compared our parallel algorithm to KaMinPar [19], a parallel multilevel framework specifically tailored for partitioning graphs into a large number of blocks. For block sizes that only contain up to 8 nodes, our best configuration produces better solutions than KaMinPar on 70% of our benchmark instances, and is on average 4 times faster. Moreover, for block sizes up to 32 our algorithm is faster by a factor of 1.5 with comparable

solution quality. Our configuration that only uses Label Propagation as local search algorithm produced better solutions than KaMinPar for block sizes up to 4 and comparable solutions for block sizes up to 16, but is on average an order of magnitude faster than KaMinPar. However, for larger block sizes (larger than 64 for FPR, 32 for LP) KaMinPar significantly outperforms our algorithms in terms of quality.

6.1 Future Work

In the future, we would like to evaluate further refinement techniques for large k. In particular, one could implement other refinement algorithm based on Integer Linear Programming (ILP) inspired by [22]. Moreover, efficient implementations of Fiduccia-Mattheyses [15] for very large k can be explored. Finally, we plan to integrate our best algorithm into the graph partitioning framework KaMinPar[3], so that KaMinPar can switch to the simpler non-multilevel algorithm for very large k.

[3] https://github.com/KaHIP/KaMinPar

References

[1] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[2] U. o. M. Laboratory of Web Algorithms. Datasets. URL:http://law.di.unimi.it/datasets.php.

[3] H. Abelson and A. A. diSessa. Turtle Geometry, The Computer as a Medium for Exploring Mathematics. *Cambridge, Massachusetts: The MIT Press*, 1984.

[4] L. Almeida and A. A. Lopes. An Ultra-Fast Modularity-Based Graph Clustering Algorithm. 2009.

[5] C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration, the VLSI Journal, 19(1-2):1–81*, 1995.

[6] T. Alsop. Fastest supercomputers by number of computer cores 2020. https://www.statista.com/statistics/268280/number-of-computer-cores-in-selected-supercomputers-worldwide/, 2021.

[7] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering. 2012.

[8] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letter*, pages 153–159, 1992.

[9] W. Chen, Y. Song, H. Bai, C. Lin, and E. Y. Chang. Parallel Spectral Clustering in Distributed Systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(3):568–586, 2011.

[10] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software, 38(1):1:1–1:25.*

[11] E. D. Dolan and J. J. Moré. Benchmarking Optimization Software with Performance Profiles. *Math. Program., 91(2):201–213*, 2002.

[12] W. E. Donath and A. J. Hoffman. Algorithms for Partitioning of Graphs and Computer Logic Based on Eigenvectors of Connection Matrices. *IBM Technical Disclosure Bulletin, 15(3):938–944*, 1972.

[13] D. E. Drake and S. Hougardy. A Linear Time Approximation Algorithm for Weighted Matchings in Graphs. *ACM Transactions on Algorithms, 1:1 (2005), 107-122*, 2005.

[14] S. Dutt. New Faster Kernighan-Lin-type Graph-Partitioning Algorithms. *In 4th IEEE/ACM Conference on Computer-Aided Design, pages 370-377*, 1993.

[15] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. *In Proceedings of the 19th Conference on Design Automation, pages 175–181*, 1982.

[16] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. Communication-free massively distributed graph generation. *In IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[17] A. George and J. W. H. Liu. Computer Solution of Large Sparse Positive Definite Systems. *Prentice–Hall, Englewood Cliffs, NJ*, 1981.

[18] T. Goehring and Y. Saad. Heuristic Algorithms for Automatic Graph Partitioning. *Tech. rep., Department of Computer Science, University of Minnesota, Minneapolis*, 1994.

[19] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier. Deep Multilevel Graph Partitioning. *CoRR*, abs/2105.02022, 2021.

[20] B. Hendrickson and R. Leland. A Multi-Level Algorithm For Partitioning Graphs. *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, pp. 28-28*, 1995.

[21] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16, 03 1995.

[22] A. Henzinger, A. Noe, and C. Schulz. ILP-based Local Search for Graph Partitioning. *CoRR*, abs/1802.07144, 2018.

[23] T. Heuer. Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning. *Bachelor Thesis, Karlsruhe Institute of Technology*, 2015.

[24] T. Heuer, P. Sanders, and S. Schlag. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[25] L. Hyafil and R. Rivest. Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems. *Technical Report 33, IRIA - Laboratoire de Recherche en Informatique et Automatique*, 1973.

[26] L. Hyafil and R. Rivest. A Simple Approximation Algorithm for the Weighted Matching Problem. *Information Processing Letter 85 (2003), 211-213*, 2002.

[27] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing, 20(1):359–392*, 1998.

[28] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal, 49(1):291–307*, 1970.

[29] F. Khorasani, R. Gupta, L. N. Bhuyan, C. Schulz, D. Strash, and M. von Looz. Scalable SIMD-Efficient Graph Processing on GPUs. *In Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques, PACT '15, pages 39–50.*

[30] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis. Improving Graph Partitioning for Modern Graphs and Architectures.

[31] J. Leskovec. Stanford Network Analysis Package (SNAP).

[32] J. Maue and P. Sanders. Engineering Algorithms for Approximate Weighted Matching. In Camil Demetrescu, editor, *Experimental Algorithms*, pages 242–255, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[33] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-Constrained Clustering. *In Experimental Algorithms, volume 8504 of LNCS, pages 351–363. Springer*, 2014.

[34] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers and Operations Research. 24 (11):*, page 1097–1100, 1997.

[35] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *Journal of Experimental Algorithmics (JEA)*, 11(2006), 2007.

[36] M. E. J. Newman. Community Detection and Graph Partitioningn. *CoRR, abs/1305.4974*, 2013.

[37] B. Peng, L. Zhang, and D. Zhang. A survey of Graph Theoretical Approaches to Image Segmentation. *Pattern Recognition 46(3):1020 – 1038*, 2013.

[38] J. R. Pilkington and S. B. Baden. Partitioning with Space-filling Curves. *Technical Report CS94-349, UC San Diego, Dept. of Computer Science and Engr.*, 1994.

[39] U. N. Raghavan, R. Albert, and S. Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E, 76(3)*, 2007.

[40] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Symposium on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

[41] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. *In The Sourcebook of Parallel Computing, pages 491–541*, 2003.

[42] C. Schulz. High quality graph partitioning. *PhD thesis*, 2012.

[43] N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei. The DAC 2012 Routability-driven Placement Contest and Benchmark Suite. *In 49th Design Automation Conference, (DAC), pages 774–782.*

[44] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. *In SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pages 1–12, doi:10.1145/1362622.1362674.*

A Detailed Performance Profiles Initial Partitioning

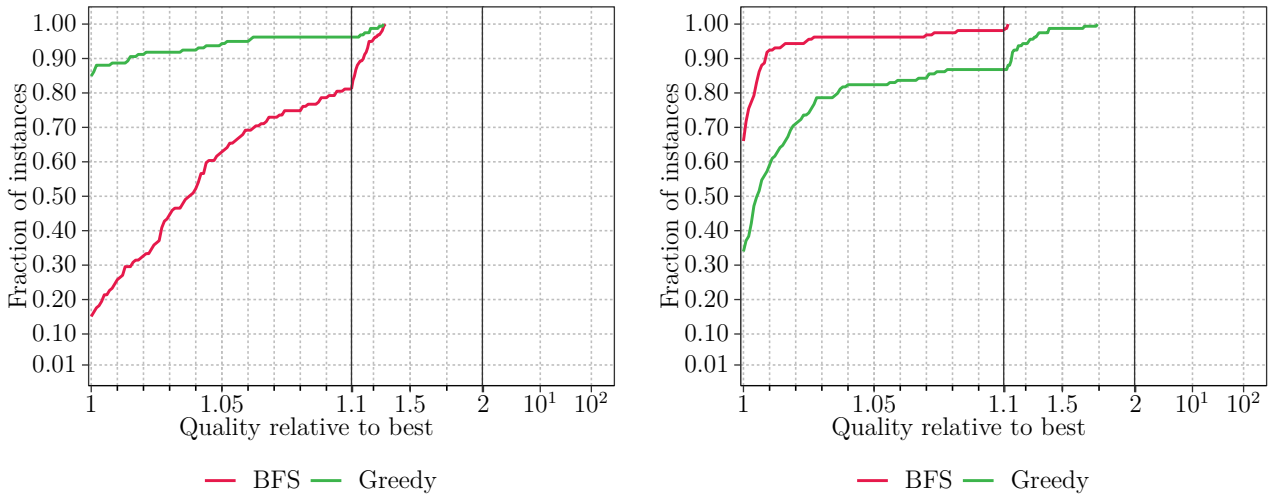


Figure 21: BFS vs. Greedy with initial partition cut (left) and cut after refinement with BPR (right).

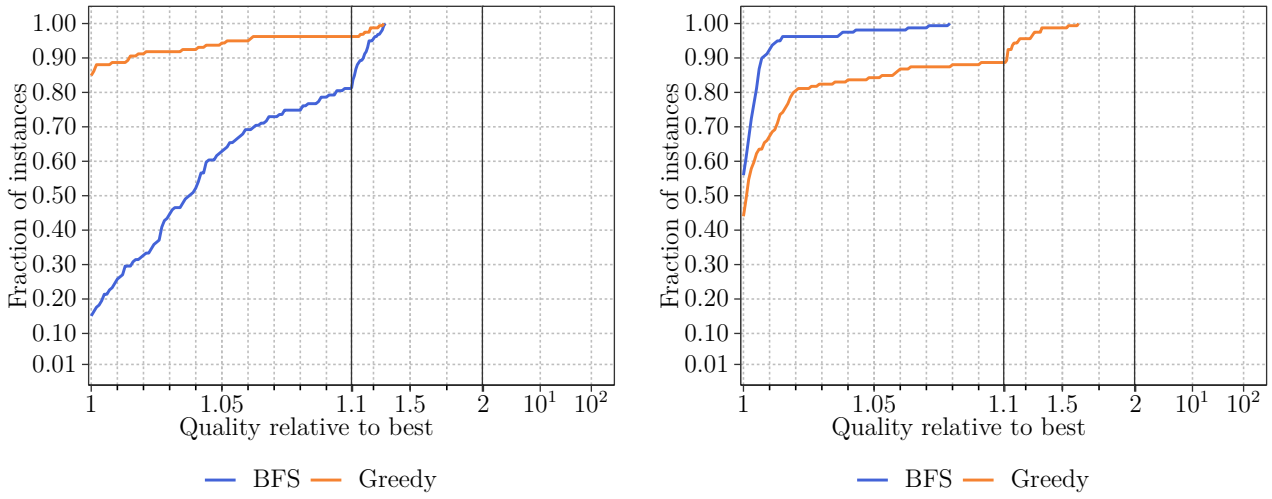


Figure 22: BFS vs. Greedy with initial partition cut (left) and cut after refinement with FPR (right).

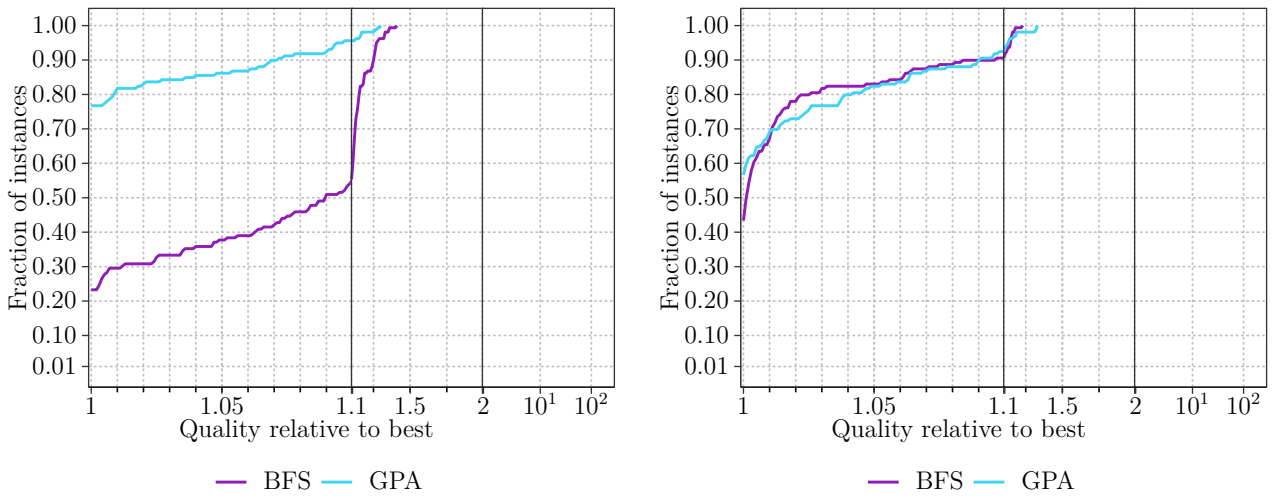


Figure 23: BFS vs. GPA (2-hop) with initial partition cut (left) and cut after refinement with Label Propagation (right).

B Quality Loss with Parallel Implementation

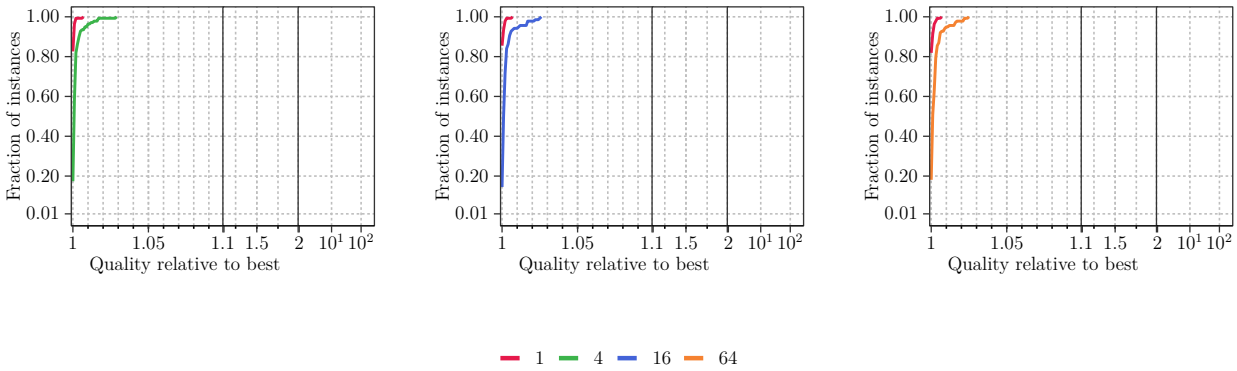


Figure 24: Quality of parallel Label Propagation relative to single-threaded execution.

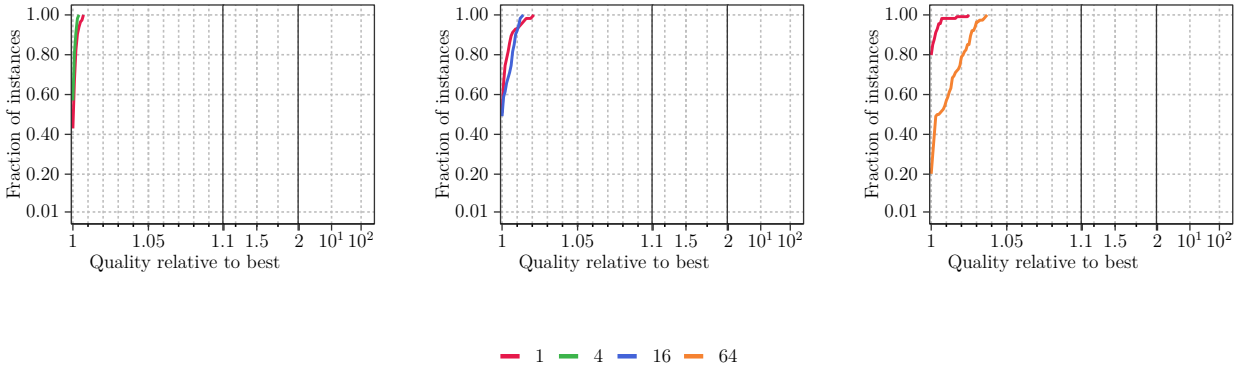


Figure 25: Quality of parallel Path Refiner relative to single-threaded execution.