Engineering a Multilevel Memetic GPU-Graph Partitioner

Jacob Rose

August 23, 2026

4213084

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Henning Woydt

Acknowledgments

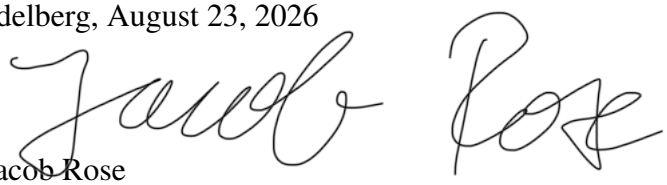
I would first like to express my sincere gratitude to Professor Christian Schulz for introducing me to the field of algorithms. His lectures were always engaging and inspiring, and played a major role in sparking my interest in this area. I am also very grateful for the opportunity to work on such an exciting and challenging research topic under his guidance. I am equally grateful to my supervisor, Henning Woydt. Throughout this project, he dedicated an incredible amount of time to our weekly meetings, always providing valuable feedback, guidance, and encouragement. These discussions were instrumental in helping me make steady progress and overcome the many obstacles during the course of this thesis. Beyond his expertise, I greatly appreciated his kindness and patience. Working with Henning has been a genuine pleasure, and I could not have wished for a better supervisor

This work was performed on the computational resource bwUniCluster funded by the Ministry of Science, Research and the Arts Baden-Württemberg and the Universities of the State of Baden-Württemberg, Germany, within the framework program bwHPC. I hereby acknowledge support by the state of Baden-Württemberg through bwHPC.

Beyond academia, I would also like to thank my family and friends for their unwavering support. I am especially grateful to my parents, who have encouraged and supported me throughout my studies in countless ways. I would also like to thank my brother, Daniel. His advice and perspective have been invaluable and have greatly enriched my studies. Many thanks to my friend Eric for making the past semesters so enjoyable. I truly appreciated sharing our enthusiasm for algorithms. Finally, I would like to express my heartfelt gratitude to the German Academic Scholarship Foundation (Studienstiftung des deutschen Volkes) for supporting my studies. Beyond the financial support, I am especially grateful for the many inspiring events that I was fortunate to be part of.

I hereby confirm, Jacob Rose, Matr. No. 4213084, that this thesis has been written independently and without unauthorised external assistance. I have not used any sources or aids other than those specified, and I have identified all passages that have been taken verbatim or paraphrased from published or unpublished works, including those from digital or AI-based sources. I confirm that any use of AI tools was discussed in advance with the supervisor of the thesis and that the agreed rules were followed. I take full responsibility for the academic quality and content of this paper, the methodology chosen and the process of its creation, as well as the literature cited. I agree that my thesis will be checked for plagiarism and possible misuse of automated text and code generation as part of university procedures.

Heidelberg, August 23, 2026



Jacob Rose

Abstract

The Graph Partitioning Problem seeks to partition a graph into k balanced blocks while minimizing inter-block edges. Owing to its numerous applications, there is a strong demand for fast and high-quality partitioning algorithms. Despite the recent succes of GPU-computing, state-of-the-art GPU graph partitioners still fall short of the solution quality achieved by their CPU counterparts. To narrow this gap, we propose MEMHEIPA, the first GPU-based multilevel memetic graph partitioner. Our algorithm maintains a population of candidate partitions throughout the multilevel hierarchy, which is iteratively improved by evolutionary operators. A crossover operator preserves promising vertex groupings shared among parent solutions, while a mutation operator based on iterated multilevel schemes further improves high-quality individuals. Additionaly, we propose the first hybrid CPU+GPU partitioner which is able to fully utilize heterogenous machines. To this end, we extend the distributed evolutionary algorithm KAFFPAE with MEMHEIPA. Extensive experiments demonstrate that MEMHEIPA substantially improves the solution quality of GPU graph partitioners. Our *strong* configuration achieves edge cuts that are on average 4.5% lower than those of the current state-of-the-art GPU partitioner, while the *fast* configuration outperforms the widely used CPU partitioner METIS in both execution time and solution quality. The hybrid algorithm KAFFPAE+GPU substantially accelerates convergence and consistently computes higher-quality partitions than the original CPU implementation in the same timeframe.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The graph partitioning problem (GPP) aims to divide the vertices of a graph into k roughly equally sized blocks while minimizing the number of edges running between different blocks. Graph partitioning has many practical applications. One of the most important application areas is parallel computing, where graph partitioning is used to balance the computational workload across multiple processing elements (PEs) while simultaneously reducing communication overhead. Many scientific simulations operate on large two- or three-dimensional computational meshes, where computation is performed on mesh elements and information is exchanged between neighboring elements. By partitioning the mesh into equally sized regions and assigning each region to a different processing element, the workload can be distributed efficiently. Minimizing the number of edges between different regions reduces the amount of communication required between processors, which is crucial for achieving high parallel efficiency [1]. Another important application of graph partitioning arises in the design of very large-scale integrated (VLSI) circuits. Here, partitioning is used to decompose large circuits into smaller submodules, thereby reducing design complexity and simplifying the design process [2]. Graph partitioning is also used in route planning algorithms. For example, the *arc flags algorithm* employs graph partitioning during a preprocessing phase to reduce the search space explored by Dijkstra's algorithm during shortest-path queries [3]. These examples represent only a small subset of the many applications of graph partitioning. Further application areas include image segmentation, computational biology, social network analysis, and SAT solving [4].

Unfortunately, the graph partitioning problem is NP-complete [5, 6], and no constant-factor approximation algorithm exists [7]. Still, due to the many applications of graph partitioning, there is a high demand for heuristic solvers. Traditionally, these graph partitioning algorithms have been designed for execution on CPUs. Among the most well known software packages are METIS [8, 9] and KAFFPA [10]. In contrast, comparatively

little work has focused on graph partitioning on GPUs. One reason for this is that GPUs are particularly effective for highly regular computations, whereas graph algorithms typically exhibit irregular memory access patterns and irregular control flow [11]. Nevertheless, the rise of general-purpose GPU computing (GPGPU) has demonstrated that GPUs can provide substantial speedups for many computational problems [12]. Furthermore, early work by Goodarzi et al. [13] and Fagginger Auer [14] demonstrated the feasibility of GPU-based graph partitioning, and more recent approaches such as JET by Gilbert et al. [15] reported significant speedups compared to METIS. However, these GPU-based graph partitioners still generally produce partitions of lower quality than modern state-of-the-art CPU-based approaches. It therefore remains an important open challenge to develop a GPU partitioning algorithm which closes this gap in partition quality, while still remaining faster than CPU partitioners. Furthermore, current approaches typically focus either on CPUs or GPUs and therefore do not fully exploit the capabilities of heterogeneous systems. Since many modern high-performance computing systems consist of both CPUs and GPUs [16], hybrid graph partitioning algorithms that efficiently utilize the entire hardware represent a promising direction for computing high-quality partitions within very short execution times.

1.2 Our Contribution

In this work, we propose MEMHEIPA (Memetic Heidelberg Partitioning), a GPU-based memetic multilevel graph partitioner that computes partitions with significantly better solution quality than existing state-of-the-art GPU algorithms. MEMHEIPA maintains a population of candidate solutions throughout the multilevel hierarchy and improves this population during each uncoarsening step, using crossover and mutation operators. The crossover identifies groups of vertices that are grouped together in all parents and transfers them to the offspring. The mutation is based on the concept of iterated multilevel schemes and guarantees non-decreasing solution quality. Through a detailed tuning study, we shed light on the effects of each hyperparameter and derive two preconfigurations *fast* and *strong*. Extensive experiments show, that both our preconfigurations outperforms the current state-of-the-art GPU partitioner JET, with *strong* achieving on average 4.5% lower edge cuts. Furthermore, our *fast* preconfiguration outperforms the well-established CPU algorithm METIS in both execution time and solution quality.

Additionaly, we propose a hybrid CPU+GPU partitioner, designed to fully exploit the resources of modern supercomputers. To this end, we extend the distributed evolutionary CPU partitioner KAFFPAE with GPU acceleration, using MEMHEIPA. Experiments demonstrate that the hybrid approach substantially accelerates convergence and produces higher-quality partitions than the original CPU implementation in the same timeframe.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the necessary preliminaries, including graph-theoretic notation, the mathematical formulation of GPP, an overview of evolutionary algorithms, and the Kokkos programming model. An overview of related work is provided in Chapter 3. We first review multilevel graph partitioning before discussing GPU-based approaches and evolutionary graph partitioning algorithms in greater detail. Our proposed algorithm, MEMHEIPA, is presented in Chapter 4. The chapter describes the design of the individual multilevel components as well as the memetic operators. Building upon this algorithm, Chapter 5 presents a hybrid CPU–GPU memetic graph partitioner that fully exploits the computational resources of heterogeneous systems. To this end, we first review the CPU-based algorithm KAFFPAE and then describe how MEMHEIPA is integrated into its memetic framework. The experimental evaluation is presented in Chapter 6. It comprises a parameter tuning study, comparisons with state-of-the-art GPU and CPU graph partitioners, and an evaluation of KAFFPAE with and without MEMHEIPA. Finally, Chapter 7 summarizes the main findings of this thesis and discusses directions for future work.

CHAPTER 2

Preliminaries

We will now present the relevant definitions which are used throughout this thesis. The chapter is structured as follows: We start by giving general definitions in Section 2.1. Then, Section 2.2 gives an introduction to graph partitioning, which is the focus of this work. Next, Section 2.3 gives an overview of evolutionary and memetic algorithms. Lastly, Section 2.4 finishes with an overview of the Kokkos framework.

2.1 General Definitions

A graph G is defined as a pair (V, E) where V denotes the set of $vertices$, and E denotes the set of $edges$. A directed edge $e = (u, v)$ connects the vertex u with vertex v. An undirected edge $e = \{u, v\}$ implies the existance of both (u, v) and (v, u). We define the size of the vertex set V as both $|V|$ and n. Similarly we define the size of the edge set E as both $|E|$ and m. A graph can further be described by a vertex-weight function $c : V \to \mathbb{R}_{>0}$ and an edge-weight function $\omega : E \to \mathbb{R}_{>0}$. We generalize these functions to sets such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. The neighborhood of a vertex v is defined as $N(v) = \{u : (v, u) \in E\}$ and its degree is defined as $d(v) = |N(v)|$. The maximum degree in the graph is denoted by $\Delta(G)$. A *matching* $M \subseteq E$ is a set of edges where no two edges share a common vertex. In other words, the graph (V, M) has maximum degree one. A matching is called *maximal* if no additional edge from E can be added without violating the matching property. It is called *maximum-weight* if its total weight $\omega(M)$ is maximum among all matchings in the graph.

2.2 Graph Partitioning and Clustering

For an undirected Graph G and an integer k, the $Graph\ Partitioning\ Problem\ (GPP)$ asks for a division of the vertex-set into k distinct $blocks$ $V_1, V_2, \ldots, V_k$. More formally,

a *k-way partition* of V can be defined as a mapping $\Pi : n \to k$ which distributes the vertices of G among k different subsets, such that $V = V_1 \cup V_2 \cup \cdots \cup V_k$ and $V_i \cap V_j = \emptyset$ for all $i \neq j$. The value $\Pi[v]$ represents the block-ID of vertex v and $\{V_1, ..., V_k\}$ the *partition* of G. A balance constraint demands that all blocks are of roughly equal weight. More precisely, we call a k-way partition ϵ-*balanced* if each block V_i satisfies the balance constraint $c(V_i) \leq L_{max} = (1 + \epsilon) \cdot \lceil \frac{c(V)}{k} \rceil, \forall\, i \in \{1 \dots k\}$ for some imbalance parameter $\epsilon \geq 0$. Typically GPP seeks to minimize the *edge cut*. An edge $e = \{u, v\}$ is considered a *cut edge* if $u \in V_i$, $v \in V_j$ and $i \neq j$. The set of all cut edges between two blocks is defined as $E_{i,j} := \{\{u, v\} \in E : u \in V_i, v \in V_j, i \neq j\}$. Then, the *edge cut* is defined as

$$\sum_{i<j} \omega(E_{i,j})$$

It has been shown that finding a partitioning with the minimum edge cut is an NP-complete problem [5, 6] , which is why in practice most solvers make use of heuristics.

The problem of *graph clustering* is closely related to graph partitioning. One also seeks to find a partition of the vertex set into blocks $C_1, ..., C_k$. However, in graph clustering, k is typically not known in advance and there is also no balance constraint placed on the size of the blocks. The set $\{C_1, ..., C_k\}$ is called a *clustering* of a graph.

2.3 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are inspired by the principles of natural evolution and model the way species adapt over time. In nature, individuals that are better adapted to their environment are more likely to survive and reproduce, thereby passing favorable traits on to the next generation. Random mutations introduce variation among offspring and allow new traits to emerge. Over many generations, this process of selection and variation leads to increasingly well-adapted populations. The effectiveness of evolution in nature is reflected in the vast diversity of species, each specialized for its own ecological niche.

Evolutionary algorithms transfer these principles to the domain of optimization. Instead of biological organisms, the algorithm maintains a population of candidate solutions, called individuals. The initial population is usually denoted with $POP_0 = \{I_1, \dots, I_{|POP|}\}$. Each individual is evaluated using a fitness function $f(\cdot)$ which measures the quality of the corresponding solution. The objective of the algorithm is usually to produce solutions which maximize the fitness function. The overall workflow of an evolutionary algorithm is illustrated in Figure 2.1. After initialization, the evolution from one generation POP_i to the next generation POP_{i+1} typically consists of the following steps:

1. A set of p parents $\{P_1, \dots, P_p\}$ is selected from the current population. Parent selection usually favors individuals with better fitness values. One commonly used selection strategy is *tournament selection*, where λ individuals are chosen uniformly

at random and the fittest among them is selected as a parent. Repeating this process p times yields the complete parent set.

2. New offspring are generated from the selected parents using variation operators such as crossover and mutation. Crossover combines information from multiple parents to create a new individual, whereas mutation introduces random modifications to a single parent.

3. The next population is formed by selecting a subset of individuals from the current population and the newly created offspring. This selection process again typically favors fitter individuals, thereby improving the overall solution quality over time.

These steps are repeated until a stopping criterion is reached, for example a time limit or a fixed number of generations. Evolutionary algorithms are often highly effective because they combine exploration of the search space through crossover and mutation with exploitation through fitness-based selection pressure [17].

The origins of evolutionary algorithms go back as early as the 1940s [18]. Since then, many different strands of this paradigm have evolved, with one of them being memetic algorithms [19]. These algorithms further include domain knowledge into the evolutionary process, leading to specialized crossover and / or mutation operators, coupled with local search techniques. One major challange for memetic algorithms in particular is early convergence, where the diversity of the population is lost too quickly and the algorithm gets stuck in a local optimum. This raises the importance for explicit strategies to enhance diversity in the population [17].

2.4 Kokkos

Since our implementation is based on *Kokkos* [20], we briefly introduce the framework here. The Kokkos ecosystem was designed to enable *performance portability*, i.e., the ability to write a single code that can efficiently execute on a wide range of hardware architectures, with performance comparable to architecture-specific implementations. Parallel algorithms can be expressed in kokkos using the three fundamental primites *parallel for*, *parallel reduce* and *parallel scan*. The parallel for simply splits all N loop iterations among the threads. Parallel reduce lets the threads collectively compute a reduction $y = \oplus_{j \in 1...N} f(j)$ for an associative operator $\oplus$ and a user-defined function $f(\cdot)$. A parallel scan produces a set of values $\{y_1, ..., y_N\}$ where each value y_i is equal to $y_i = \oplus_{j \in 1...i} f(j)$.

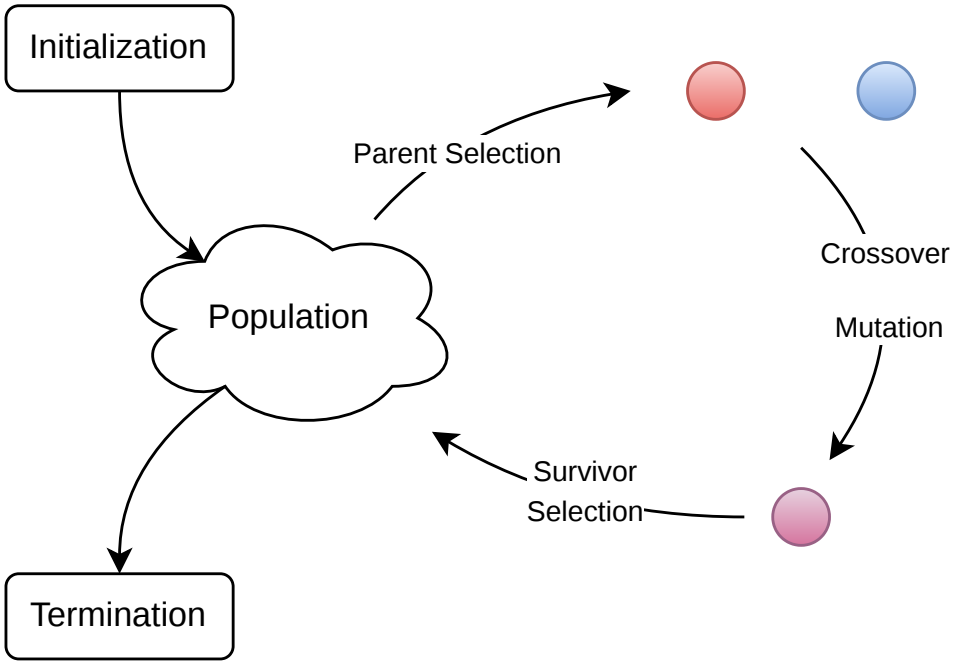


Figure 2.1: General structure of an evolutionary algorithm. Adapted from [17](p. 27).

CHAPTER 3

Related Work

Graph partitioning is a well-studied research area with an extensive body of literature. For a comprehensive overview, we refer the reader to the surveys in [4, 21]. This chapter focuses on the topics most relevant to our work, namely multilevel graph partitioning, GPU graph partitioning, and evolutionary graph partitioning. Finally, we conclude with a brief overview of the Walshaw benchmark.

3.1 Multilevel Graph Partitioning

Since the graph partitioning problem is NP-complete, heuristics are used in practice. The multilevel graph partitioning (MGP) approach counts as one of the most effective heuristics, since it is able to quickly compute high quality partitions. Multilevel schemes in general have their origin in the context of linear systems of equations [22] and have been introduced to the GPP by Barnard and Simon [23].

The general structure of multilevel schemes is to construct a hierarchy of increasingly coarser approximations of the graph, compute a partition on the smallest graph, and then successively project and refine the partition back to the original graph. Coarser graphs are created such that they retain the structure of the original graph, allowing a high-quality partition on the coarsest graph to be translated into a strong partition on the original graph. MGP consist of three main phases: *coarsening*, *initial partitioning* and *uncoarsening*. We describe each phase in more detail below.

Coarsening During coarsening the size of the input graph is progressively reduced and a hierarchy of coarser graphs is created. One such coarsening step involves grouping together vertices and then contracting each group of vertices into a new vertex of the coarser graph. The two most common ways to create vertex-groups make use of *matchings* or *clusterings*. In the case of matchings, two matched vertices u and v are contracted into a new vertex x. Their vertex-weights are summed, such that $c(x) = c(u) + c(v)$ and the new

vertex x is connected to both $N(u)$ and $N(v)$. If a vertex y is connected to both u and v, a single edge will be created with $\omega(\{x, y\}) = \omega(\{u, y\}) + \omega(\{v, y\})$. These coarsening steps are repeated until the size of the graph falls below a certain threshold, usually $c \cdot k$ with c being a hyperparameter.

One popular heuristic for computing a matching is the *heavy edge matching* (HEM), which aims to find a maximal matching containing edges of large weight. Contracting these heavy edges leads to a strong decrease of edge weights in the coarser graph. Since the objective of graph partitioning is to minimize the edge cut, constructing graphs with low edge weights is beneficial. To be precise, HEM processes each vertex v of the graph in a random order and attempts to match it with an unmatched neighbor u, such that the weight of the edge (v, u) is maximized [8] .

Initial Partitioning Once coarsening has finished, the final *coarsest graph* is partitioned. Since this graph is tiny, a high-quality partition can be computed cheaply. One common approach for this is *recursive bisection*. Here, the graph is first split into two parts, and then each part is again recursively bisected until a k-way partition is obtained. If k is not a power of two, the bisection is applied to only one of the two subgraphs at certain steps in order to obtain the required number of blocks. A bisection can be computed, for example using *graph growing*: Starting from a random vertex, a region is grown by performing a breadth first search (BFS) from that vertex, and adding all the discovered vertices until the weight of that region reaches half of the total graph weight. *Greedy graph growing* is a variation of this method, which is able to find lower edge cuts: Instead of using a BFS, the region is progressively grown by adding the adjacent vertex that results in the smallest increase of the edge cut. Since the quality of the solution depends on the starting vertex, these procedures are typically repeated multiple times and at the end the best partition is selected [8].

Compared to recursive bisection, there also exist methods that directly compute a k-way partition of the graph. One representative method of this class is the *bubble framework* [24]. Conceptually, the bubble framework generalizes the idea of graph growing. The main idea is to maintain k seed vertices, from which k bubbles are grown using BFS. To obtain a good partition, the algorithm alternates between two steps. First, the bubbles are grown by performing a BFS starting from each seed vertex. Second, for each bubble, a center vertex is computed, defined as the vertex that minimizes the sum of distances to all other vertices within the same bubble. These center vertices are then used as the new seed vertices in the next iteration. The algorithm terminates once the seed vertices stabilize, i.e., when the center vertices of the bubbles stop moving, or when the partition quality does not improve for a fixed number of iterations (typically 10). The resulting k bubble regions are then used as the blocks of the k-way partition.

Uncoarsening During the uncoarsening phase, the graph is successively expanded back to its original size and structure. In each uncoarsening step, previously contracted

vertices are uncontracted. Additionaly, the partition of the coarser graph is projected onto the finer graph by assigning each uncontracted vertex to the same block as its corresponding coarse vertex. However, a partition computed for the coarser graph might not be optimal for the finer graph. Therefore, *refinement* algorithms are applied after each uncoarsening step to improve the partition quality while maintaining balance constraints. Refinement algorithms are typically *local search* heuristics which move a set of vertices, each from its respective source block to another, based on the notion of a vertex's *gain*. In the context of k-way partitioning, the gain of a vertex $G_b(v)$ describes the change in edge cut resulting from moving the vertex v to a different block b.

Most refinement algorithms are based on the algorithm proposed by Kernighan-Lin [25] (referred to as *KL algorithm*) and their variants [26, 9]. Starting from an initial bisection, the KL algorithm repeatedly performs vertex swaps between the partitions, picking the pair of vertices with the highest commulative gain in each step. It terminates once a locally optimal partitioning has been obtained.

One popular variant of the KL algorithm was introduced by Fiduccia and Matthey-ses [26] (referred to as *FM algorithm*). Their algorithm was also designed for the bipartitioning problem ($k = 2$). The idea behind the algorithm is to find a sequence of vertex moves between V_1 and V_2 which maximizes the gain. In more detail, the algorithm proceeds in multiple passes, where each vertex may be moved at most once per pass. During a pass, a sequence of vertex moves is constructed by repeatedly selecting the vertex with maximum gain and moving it to the opposite block. After a move, the vertex is locked and excluded from further consideration in the current pass. The vertices are selected alternatingly from V_1 and V_2 such that any prefix of the move-sequence keeps the balance maintained. Once all vertices have been considered, the algorithm selects the prefix of the move sequence that yields the maximum cumulative gain and commits only these moves. This process is repeated until no further improvement can be achieved.

Karypis and Kumar propose an efficient extension of the FM algorithm to the k-way partitioning case (k > 2) [9]. They also perform multiple passes, with each pass performing the following steps: In the beginning, they scan all vertices and determine their maximum gain $G_{max}(v) = max\{G_b(v) | \exists u \in N(v) : \Pi[u] = b\}$. All vertices with positive maximum gain are inserted into a priority queue. The algorithm then progressively selects the vertex v with the highest gain from the priority queue and moves the vertex to the neighboring block which maximizes the gain while maintaining the balance constraint. Just like in the FM algorithm, after a vertex has been moved it is not considered any further in the current pass. The algorithm stops early when x moves have been performed which did not improve the edge cut, with x being a hyperparameter. These negative moves are then undone.

Refinement algorithms have also taken inspiration from *tabu search* [27]. The main

concept in tabu search is the use of *adaptive memory* to store the search history. This information can be used to prevent the algorithm from cycling, i.e. repeatedly executing the same set of moves. A common application of tabu search in the context of k-way partitioning is to maintain a tabu list which stores the vertex moves which were performed last. Instead of locking vertices for a whole pass like in the FM algorithms, all the vertices in the tabu list are prohibited from moving back to their source partition for only a specific number of steps, called the tabu tenure [28, 29].

A greedy variant of the FM algorithm was introduced by Karypis and Kumar [30]. In contrast to the classical FM algorithm, all moves are committed irreversibly and there is no final selection of the best move-seqeuence. Additionaly, only positive gain moves are allowed. These changes make the method faster, but less capable of escaping local minima. To be precise, the algorithm proceeds in multiple iterations and in each iteration, all vertices are visited in some (possibly random) order. If a vertex has positive gain, it is immediately moved to the best adjacent block, which does not violate the balance constraint. An iteration terminates once all vertices have been processed.

The same algorithmic idea has been previously proposed in the context of detecting communities in networks, by the *label propagation* algorithm [31]. At the beginning of the algorithm, each vertex is assigned a unique label. Subsequently, the algorithm performs rounds where all vertices are visited in a random order. When a vertex is visited, it is assigned the label held by the majority of its neighbors. The main difference to the greedy FM algorithm is the initialization of vertices and that no balance constraint is imposed. Consequently, at the end of label propagation all vertices may also hold the same label or completely different labels.

3.2 GPU Graph Partitioning

GPUs have become increasingly important in high-performance computing due to their massive computational throughput. They are especially effective for regular workloads with predictable memory access patterns and structured computation. Graph algorithms, however, are often inherently irregular. Their sparse and data-dependent structure can lead to irregular memory accesses, load imbalance, and unpredictable control flow, making it difficult to fully utilize the parallel capabilities of modern GPUs. As a result, designing efficient GPU implementations for graph problems remains a challenging task. Nevertheless, a growing body of research has demonstrated that GPUs can also be successfully applied to graph partitioning. In the following, we review several GPU-based graph partitioning approaches. This chapter begins with a historical overview of GPU-based graph partitioning approaches which discusses the early work of Fagginger Auer and Bisseling [14], Goodarzi et al. [13, 32] and Wan-Luan Lee et al. [33]. The chapter ends with the work of Gilbert et al. [15]. Their algorithm JET represents the most recent and competitive GPU-based partitioner.

Fagginger Auer and Bisseling [14] were among the first to investigate multilevel graph partitioning on GPUs. They proposed two GPU-based multilevel algorithms for computing graph bipartitions, which can subsequently be used within a recursive bisection framework to obtain k-way partitions. Both algorithms use the same matching procedure during coarsening, which is iterative and continues until a maximal matching is obtained. In each iteration, unmatched vertices are first colored either blue or red. Afterwards, every blue vertex proposes to a red neighboring vertex it wants to match with. The red vertices then respond to one of the received proposals. Finally, all vertex pairs with compatible proposals and responses are matched. The two partitioning algorithms mainly differ in their initial partitioning and refinement strategies. Their first approach computes the initial partition using greedy graph growing and applies an FM-style local search algorithm for refinement. Their second approach is based on spectral graph partitioning. Spectral methods use linear algebra techniques and operate on the Laplacian matrix of the graph. The initial partition is computed using the *Fiedler vector*, which is the eigenvector corresponding to the second-smallest eigenvalue of the Laplacian matrix. Refinement is then performed using the Jacobi–Davidson method. In their evaluation, the authors compare their algorithms against METIS [8] and show that the greedy variant achieves comparable partition quality. Moreover, for very large graphs containing millions of vertices, their GPU implementation outperforms METIS in terms of running time. Their work demonstrates that GPUs constitute a viable alternative to traditional CPU-based graph partitioners, particularly for large-scale instances.

Goodarzi et al. [13] present a lock-free multilevel graph partitioner for heterogeneous CPU-GPU architectures. Their approach starts with coarsening the graph on the GPU, for which they employ a two-phase heavy edge matching scheme. At the beginning of this scheme, each vertex selects its preferred matching partner. Then, in a second step, a match is created only if both vertices mutually select each other. Otherwise the vertex matches with itself. Once the graph becomes too small to exploit the massive GPU parallelism, it is transferred to the CPU, where coarsening continues until the coarsest graph is obtained. Initial partitioning is then computed on the CPU (using MT-METIS [34]). Uncoarsening starts on the CPU and is performed until the graph gets large enough again to utilize the GPU resources. At that point it is transferred back to the GPU, where the remaining uncoarsening steps are performed. The authors utilize MT-METIS [34] for coarsening, initial partitioning and uncoarsening on the CPU. Their GPU parallel refinement algorithm distributes the vertices among the threads. Each thread computes gain values for boundary vertices and commits candidate moves into buffers (one buffer per partition). Afterwards these buffers are sorted and feasible moves, which maintain the balance constraint are selected. To avoid conflicting concurrent exchanges between neighboring partitions, each refinement step is split into two iterations where vertices can only move in one direction. The authors report speedups ranging from 1.5x to 5x over METIS while achieving comparable partition quality.

Later, they propose several hardware-specific improvements in [32]. The coarsening phase continues to use HEM, but is extended with several GPU-specific optimizations. They mitigate transfering data between the CPU and GPU, by executing the complete graph partitioner on the GPU. For this, they replace the initial partitioning on the CPU with a graph growing technique, using a parallel breadth first search (BFS). They also change their refinement to allow groups of threads to collaborate. As before, the method begins with each thread computing the gain and destination partition for their vertex. However all moves are commited to one global buffer. The moves in this buffer are now explored collectively. First the ℓ best gain moves are determined using a parallel reduction, with ℓ being a hyperparameter. Since the gain values were computed in parallel, some moves might interfere with each other. To find the most beneficial set of moves, all 2^ℓ combinations of these ℓ moves are evaluated by 2^ℓ groups of threads. The combination with the highest gain is finally performed. These two steps are repeated until the buffer is empty. Their improved algorithm is able to compute partitions of similar quality to MT-METIS [34] while on average being 1.9 times faster.

Wan-Luan Lee et al. [33] present G-KWAY, a multilevel GPU graph partitioner that improves upon the approach of Godarzi et al. [32] by exposing additional parallelism during both coarsening and refinement. The first of their key contributions is a novel union-find based coarsening algorithm that reduces the number of coarsening levels. Rather than matching only pairs of vertices as in the classical HEM algorithm, their approach merges vertices into larger subsets, which are subsequently contracted into vertices of the coarser graph. More precisely, each vertex u first selects a neighboring vertex v which maximizes the score $s(u, v) = c \cdot \omega(u, v) - d(v)$, where $c \geq \Delta$. Afterwards, the subsets are created by merging each vertex with their selected neighbor. Once the number of vertices falls below $\frac{|V|}{20 \cdot \log_2(k)}$, the graph is transferred to the CPU where an initial partition is computed using METIS [8]. Afterwards it is copied back to the GPU, where uncoarsening and refinement are performed. The second key contribution is an independent set-based refinement algorithm designed to maximize parallelism while preventing neighboring vertex moves from interfering. Initially, the best destination block is determined for every boundary vertex based on its gain. An independent set of vertices to move is then determined. These vertex moves are subsequently sorted in descending order of gain, thereby yielding a move sequence. Finally, the longest prefix of this sequence that satisfies the balance constraint is applied. The refinement process is repeated until no further positive-gain moves can be performed. Their experiments demonstrate that G-KWAY achieves partition quality comparable to MT-METIS and the algorithm of Godarzi et al. [32], while reducing the execution time by a factor of $8.6\times$ and $3.8\times$, respectively.

3.2.1 JET

Gilbert et al. [15] present JET, a multilevel GPU graph partitioner. The algorithm extends their previous work on GPU-based coarsening and contraction algorithms [35] by a novel

refinement algorithm, to complete the multilevel scheme. The implementation is based on the *Kokkos* framework [36].

Coarsening JET uses a two-hop matcher, which was originally developed for *Mt-Metis* [34], a shared memory graph partitioner. The method begins by computing a standard heavy edge matching, which aims to match each vertex v with a neighbor u, such that the weight of the edge (v, u) is maximized. A sequential implementation visits the vertices in a random order. In the parallel setting however, the algorithm performs multiple passes: In the first pass, each vertex independently selects its preferred neighbor, i.e. the unmatched adjacent vertex connected by the heaviest edge. In the second pass, vertices concurrently attempt to match with their selected preference using atomic compare-and-swap operations to ensure that both vertices are still unmatched and can be claimed safely without locks. Successfully matched vertices are assigned the same matching ID, while unmatched vertices are collected and processed again in the next pass.

If this initial matching leaves more than 25% of vertices unmatched, the algorithm is extended to also consider pairs of vertices which have a distance of two in the graph, i.e., vertices that are *two hops* apart. However, the algorithm does not immediately consider the full set of such vertices. Instead, it first proceeds hierarchically through two smaller subsets, namely *leafs* and *twins*. Leafs are vertices with an identical neighbor, and twins are vertices that share an identical neighborhood. More specifically, the matching process proceeds in stages and begins with matching leafs. If more than 25% of vertices remain unmatched after matching the leafs, the algorithm then considers twins. If this still leaves more than 25% of vertices unmatched, it finally expands the search to the full set, referred to as *relatives*. As a result, a mapping $M : n \to n_c$ from the vertices of the finer graph to the vertices of the coarser graph is produced. This mapping is used to create the next coarser graph in the contraction step.

Contraction The main challenge in parallelizing this graph construction step is that multiple threads insert edges into the coarse graph concurrently. This requires careful handling to avoid race conditions, such as overwriting edges or computing incorrect edge weights. JET adresses this issue by using a fine grained per-vertex hashing scheme: Two hash arrays, H_v and H_w, are used to store edge endpoints and their corresponding weights. Each vertex of the coarser graph v_c is assigned a dedicated subarray within these hash tables, to insert their neighbors. To determine these subarray, the algorithm parallelizes over the number of vertices of the finer graph n, and lets each vertex atomically add an approximation of the size of their neighborhood to a helper array B at the position $M[v]$. Subsequently, an exclusive prefix sum over B is performed, yielding an offset array O, where the interval $[\, O[v_c], O[v_c + 1] \,)$ defines the index range into the hash arrays assigned to the coarse vertex v_c.

In the following step, the edges are inserted into the hash arrays. For this, the algorithm parallelizes across vertices of the finer graph and, for each vertex, across its incident edges.

A thread responsible for an edge (u, v) cycles through the subarray of the coarser vertex u_c (which is given by $M[u]$) and attempts to insert the endpoint v_c. If both vertices contracted to the same coarser vertex ($v_c = u_c$), no insertion is required. To handle concurrent insertions, threads use atomic compare-and-swap (CAS) operations: If a free slot is found, the endpoint v_c is inserted; if v_c is already present, the corresponding edge weight is updated using an atomic operation. If the current slot is occupied, it continues cycling. Because the hash arrays are sized using upper bounds, some entries may remain unused. In a final step, these empty entries are removed by extracting the actual graph structure from the hash arrays, which can be done efficiently using two parallel prefix scans.

Initial Partitioning Coarsening stops when the number of vertices in the graph has fallen below $8k$, where k is the desired number of blocks. Since this coarsest graph is too small to exploit the massive parallelism of the GPU, it is copied to the CPU for initial partitioning, for which METIS [8] is utilized. METIS employs a heavy edge matching to coarsen the graph, performs initial partitioning with a recursive bisection method, and uses a FM-style refinement in the uncoarsening phase. Afterwards, the partition is copied back to the GPU.

Uncoarsening In the uncoarsening step, the coarser graph G_{l+1} is replaced by the next finer graph G_l. This can be performed trivially by memorizing the hierarchy of graphs created during coarsening and contraction. The remaining task is to project the partition Π of the graph G_{l+1} onto G_l. This is achieved by setting $\Pi[v] = \Pi[M[V]]$ for all vertices $v \in G_l$, where M denotes the mapping which was used to contract G_l into G_{l+1}.

Refinement Parallel refinement introduces several challenges. First moving many vertices concurrently can destroy the balance, for example if all vertices decide on the same destination block. Second, even if individual moves have positive gain, simultaneous moves of adjacent vertices may interfere and degrade the overall solution. JET addresses these issues by splitting the refinement into two phases: The first phase performs an unconstrained label propagation, which may temporarily produce overweight blocks in order to expose sufficient parallelism. The second phase rebalances the partition by moving vertices out of overweight blocks while minimizing any loss in solution quality. This two-phase approach not only enables the refinement algorithm to achieve a high degree of parallelism but may also help it escape local minima.

In the unconstrained label propagation phase, each vertex v first determines its optimal destination block b based on the gain $G_b(v)$, i.e., the reduction in edge cut achieved by moving v to b. This gain is computed with respect to the current partition state only and does not account for concurrent moves of other vertices. Afterwards, two filtering steps are applied. The first filter selects vertices whose move yields either a positive gain or a slightly negative one: $G_b(v) \geq 0 \ \vee \ -G_b(v) < \lfloor c \cdot \mathrm{conn}(v, \Pi(v)) \rfloor$ where $c \in [0, 1]$ is a tuning parameter and $\mathrm{conn}(v, \Pi[v]) = \sum_{\{u | u \in N(v) \wedge \Pi[u] = \Pi[v]\}} \omega(v, u)$ describes the connectivity of

vertex v to its block $\Pi[v]$. Allowing slightly negative gains helps the algorithm escape local minima. All vertices that pass the first filter are collected in a list X

The second filter, referred to as the *afterburner*, mitigates the fact that gains were computed independently of neighboring vertex moves. To address this, the gain is recomputed for all vertices that passed the first filter, this time under an approximation of the next partition state. This approximation is based on an ordering ord of the vertices, defined as follows:

$$\begin{cases} \mathrm{ord}(u) < \mathrm{ord}(v) & \text{if } u \in X \wedge G_b(u) > G_b(v), \\ \mathrm{ord}(u) < \mathrm{ord}(v) & \text{if } u \in X \wedge G_b(u) = G_b(v) \wedge u < v, \\ \mathrm{ord}(u) > \mathrm{ord}(v) & \text{otherwise.} \end{cases}$$

When evaluating a vertex, neighbors with a lower order are assumed to have already moved, while others are assumed to remain in place. Only vertices whose gain remains positive after this second filtering step are finally moved.

Since the unconstrained label propagation can create overweight blocks, a way to restore balance afterwards is crucial. JET introduces the two methods *weak rebalancing* and *strong rebalancing* for this purpose. *Weak rebalancing* starts by computing the maximum gain (i.e. the minimum loss) for all vertices in the set of overweight blocks A . These vertices can pick from a set of candidate blocks B, which have a weight smaller than σ. The value $\sigma \leq L_{max}$ is used to filter blocks which are able to accept multiple vertices before becoming overweight. Preferably the nodes should move to adjacent blocks, but a random block is picked, if no adjacent blocks is eligible (size smaller σ). Next, for each overweight block, the vertices are approximately sorted in terms of increasing loss (decreasing gain). Then the minimum prefix L'_x of this approximately sorted sequence L' is choosen, which will restore the balance of this block.

Approximately sorting the vertices of a single overweight block works using hash-buckets. Each vertex determines the bucket to add himself to using the following formula:

$$slot(G_b(v)) = \begin{cases} 2 + \lfloor \log_2(x) \rfloor & \text{if } G_b(v) < 0, \\ 1 & \text{if } G_b(v) = 0, \\ 0 & \text{if } G_b(v) > 0. \end{cases}$$

Furthermore, each bucket is subdivided into ρ buckets to reduce atomic contention for the size counters of each bucket. For a vertex assigned to bucket x, the target sub-bucket is determined by the value $v \bmod \rho$. Finally, the prefix L'_x is drawn from the buckets, starting from the buckets with a smaller ID.

Weak rebalancing is designed to optimize the loss, but it can take many iterations until the graph becomes balanced, because the destination partitions can also become overloaded during rebalancing. In contrast, *strong rebalancing* is typically able to restore balance within a single iteration, but generally leads to a larger increase in edge cut. The algorithm follows the same general structure as *weak rebalancing*, but uses a different loss definition

and explicitly seeks to avoid overloading destination blocks. Instead of using the maximum gain of a vertex, *strong rebalancing* uses the mean gain

$$\text{mean}_{i \in U \cap A_v}(\ \text{conn}(v, i)\ ) - \text{conn}(v, \Pi[v]),$$

where

$$A_v = \{i \mid \exists u \in N(v) : \Pi[u] = i\}$$

denotes the set of indices of adjacent blocks of v, and

$$U = \{i : |V_i| \leq \sigma\}$$

is the set of indices of eligible underloaded blocks. Here

$$\text{conn}(v, i) = \sum_{\{u | u \in N(v) \wedge \Pi[u] = i\}} \omega(v, u)$$

describes the connectivity of vertex v to block V_i.

As in *weak rebalancing*, the vertices of each overloaded block are approximately sorted according to their gain values and the minimum prefix restoring balance is selected. When assigning the evicted vertices to destination blocks, *strong rebalancing* employs an additional mechanism intended to prevent destination blocks from becoming overloaded. Each underloaded block $b \in B$ tries to select close to $\sigma - |b|$ of the evicted vertices to include in its set. As a result, the assignment is guided primarily by the balancing constraints of the blocks, rather than by connectivity to the neighboring vertices, potentially resulting in a larger increase in edge cut than *weak rebalancing*. Further, note that this mechanism does not guarantee that no new overloaded blocks are created. However, in practice only a few iterations are typically required to reach a balanced partition.

3.3 Evolutionary Graph Partitioning

There has been a lot of research on evolutionary algorithms for graph partitioning. We refer the reader to the survey by Kim et al. [37] for a thorough study and focus here on presenting evolutionary algorithms which work with the multilevel scheme.

Soper et al. [38] were the first to combine evolutionary search techniques with multilevel graph partitioning. A key component of their approach is the use of *biases*, which are additional vertex and edge weights that influence the behavior of the multilevel partitioner. Offspring partitions are created by adding vertex and edge biases to the input graph and subsequently applying the multilevel partitioner *Jostle* [39]. The bias of an edge (u, v) is computed as the sum of the biases of u and v, plus one. These edge biases are then added to the original edge weights, thereby influencing both the contraction and refinement phases of *Jostle*. During contraction, edges with higher weights are more likely to be contracted, while during refinement, vertex gains are computed using the modified edge weights. In both crossover and mutation, vertices near cut boundaries receive low biases, which guides the partitioner towards similar cut regions in the offspring. The algorithm is able to produce high quality partitions, but at the expense of running times of up to one week.

3.3.1 Multilevel Memetic Graph Partitioning

Benlic et al. [40] introduced a multilevel memetic algorithm (MMA) designed for the perfectly balanced graph partitioning problem ($\epsilon = 0$). They integrate their memetic operators directly into the multilevel scheme and apply them to the population during each uncoarsening step. Evaluation on the well-established Walshaw benchmark demonstrated the effectiveness of their method, by its ability to recompute or improve more than two-thirds of the best-known perfectly balanced partitions (refer to Section 3.4 for details on the Walshaw benchmark).

Coarsening. The algorithm employs a heavy edge matching to contract the graph. The HEM visits all vertices in a random order and tries to match each vertex v with an unmatched neighbor u such that the weight of the edge (v, u) is maximized.

Initial Partitioning. Once the graph size falls below the coarsening threshold of 200 vertices, an initial population is created. Individuals of the population are partitions of the graph, and each individual is created by randomly assigning vertices to blocks while respecting the balance constraint. Afterwards, each individual is refined via tabu search, and lastly the memetic refinement is applied to the whole population.

Uncoarsening. In each uncoarsening step, all individuals are projected onto the next finer graph and refined using tabu search. Afterwards, the memetic refinement is applied to the entire population.

Memetic Refinement. The memetic refinement generates θ new offspring, each according to the following procedure: First, p parents are chosen using tournament selection, where the best individual from a random pool of size λ becomes a parent. Thereafter, an offspring is produced with their backbone based crossover operator. Once it has been created, it is refined via tabu search and then evaluated for inclusion in the population.

Backbone Based Crossover (BBC). The primary contribution of their work is the BBC operator, which aims at passing on promising qualities of all parents onto the offspring. The authors define the backbone as a set of k subsets of vertices which are grouped together in all the optimal partitions of the graph. However, since the optimal partitions are typically unknown, this formal definition serves a primarily theoretical purpose. To make the concept applicable, the authors utilize a relaxed backbone definition for their BBC operator, which seeks to identify k subsets of vertices which are shared across the set of locally optimal parent individuals.

Based on this relaxed backbone definition, the BBC operator constructs a new offspring by repeating the following procedure k times: Given the set of parent partitions $\{P_1, \ldots, P_p\}$, the algorithm first selects the heaviest block among all parents. More formally, let

$$m = \arg \max_{j \in \{1,\ldots,p\}} \left\{ \max_{V_i \in P_j} \omega(V_i) \right\}$$

denote the index of the parent containing the heaviest block and let

$$V^\star = \arg \max_{V_i \in P_m} \omega(V_i)$$

be this block. $V^\star$ will only be picked from P_m if P_m was chosen no more than $\lceil k/p \rceil$ times already. Otherwise, $V^\star$ is picked from another parent. In the next step, the algorithm determines, for every parent P_x with $x \neq m$, the block with maximum overlap with $V^\star$. The corresponding overlap set is defined as

$$O_x = V^\star \cap \arg\max_{V_i \in P_x} |V_i \cap V^\star|.$$

One backbone component is then computed as

$$B = \bigcap_{x \neq m} O_x,$$

i.e. the set of vertices that are grouped together across all parent individuals. This set initializes the offspring block V_t, where t denotes the current iteration of the algorithm ($V_t \leftarrow B$). Vertices contained in $V^\star \setminus B$ may still be added to V_t. A vertex $v \in V^\star \setminus B$ is inserted with probability

$$\frac{c(v)}{p-1},$$

where $c(v)$ denotes the number of overlap sets O_x containing v. Finally, all vertices assigned to V_t are removed from the parent partitions before the next iteration begins.

After these k iterations, there may be leftover vertices among the parents, i.e. vertices which were not assigned to any block of the offspring. In a final step, these vertices are distributed randomly without violating the balance constraint.

Pool Updating. To prevent premature convergence, the replacement strategy considers both partition quality (fitness) and distance between individuals (diversity). A new offspring is incorporated into the population if it improves the best solution quality or if its minimum distance to the population $D_{0,POP}$ exceeds the current minimum distance between any two existing individuals D_{min}. Distances between two k-way partitions are measured using the set-theoretic partition distance $d(\cdot,\cdot)$ [41]. Given two partitions $\{V_1,\ldots,V_k\}$ and $\{V'_1,\ldots,V'_k\}$, this measure determines the number of vertices that need to be reassigned in order to transform one partition into the other. To compute this value, a matrix $M \in \mathbb{N}^{k\times k}$ is constructed, where each entry $M_{i,j}$ denotes the size of the intersection $|V_i \cap V'_j|$. The objective is then, to find a selection of cells of M such that no row or column contains more than one cell, while maximizing the sum of the cell values. More formally, one seeks the value

$$\mathrm{sim}(\{V_1,\ldots,V_k\},\{V'_1,\ldots,V'_k\}) = \max_{\sigma\in\Psi} \sum_{i=1}^{k} M_{i,\sigma(i)}.$$

where Ψ is the set of all possible permutations of $\{1,...,k\}$. This value sim gives the similarity between both partitions and subtracting this from the size of the vertex-set gives the distance, i.e.

$$d(\{V_1,\ldots,V_k\},\{V'_1,\ldots,V'_k\}) = |V| - \mathrm{sim}(\{V_1,\ldots,V_k\},\{V'_1,\ldots,V'_k\}).$$

Given the population $POP = \{I_1, ..., I_{|POP|}\}$ the value $D_{i,POP} = min\{d(I_i, I_j) : I_j \in POP, i \neq j\}$ specifies the minimum distance between individual I_i and any other individual in the population. The Value $D_{min} = min\{D_{i,POP} : i \in 1 \dots |POP|\}$ specifies the minimum distance in the whole population. If an offspring is incorporated into the population, the individual with the worst *goodness score* $g(i) = f(I_i) + \frac{\beta}{D_{i,POP}}$ is evicted to maintain a healthy population diversity. The hyperparameter β determines how much value is given to diversity and set to $\beta = 0.08 \cdot |V|$.

3.3.2 KaFFPaE

Sanders and Schulz present KaFFPaE (Karlsruhe Fast Flow Partitioner Evolutionary) [42], a distributed evolutionary graph partitioner which utilizes the multilevel graph partitioner KaFFPa (Karlsruhe Fast Flow Partitioner). In contrast to earlier evolutionary algorithms that use a multilevel partitioner, KaFFPaE does not rely on adding biases to the edge weights [38]. Instead, KaFFPaE introduces novel mutation and crossover operators that guide the multilevel solver. We start by giving an overview of KaFFPaE and then elucidate KaFFPa, as well as the recombination and mutation operators.

Algorithm Overview. The distributed algorithm is parallelized across a set of processing elements (PEs), where each PE independently performs the following operations with different random seeds: First, an initial population is constructed. To this end, each PE is assigned a time budget of $t_{init} = t_{total}/f$, where f is a hyperparameter and t_{total} denotes the total runtime given to KaFFPaE. After measuring the average runtime $\bar{t}$ of a single partitioning call, the local population size is determined as $S = \lceil t_{init}/\bar{t} \rceil$. Then, each PE initializes its population by generating S individuals using KaFFPa. After initialization, each PE iteratively executes rounds until the global time limit is reached. Each round consists of a local partitioning phase followed by a communication phase. During the local partitioning phase, new individuals are generated either via mutation or recombination operators, with the operator being selected at random. The authors recommend a mutation-to-recombination ratio of 1:9. Whenever a new individual is generated, it is incorporated into the population provided that the population contains individuals of worse or equal quality. In this case, the individual that is most similar to the newly generated individual is evicted. The communication phase is organized in $log\, p$ rounds where p denotes the number of PEs. In each round of the communication phase, the current best solution of a PE is sent to a randomly selected PE from a set of eligible PEs. Incoming partitions are received asynchronously and incorporated into the local population using the eviction strategy described above. Communication is non-blocking, i.e. a PE does not wait for acknowledgements or incoming messages before proceeding with its computation. After the time limit is reached, the best solution among all PEs is collected and returned.

KaFFPa. We now explain the multilevel algorithm which KaFFPaE uses to create individuals. KaFFPa employs the global path algorithm (GPA) [43] to compute a maximum weight matching, during the coarsening phase. The idea of the algorithm is to first grow

paths and even length cycles and then extract a matching from these paths. More precisely, the algorithm starts with all vertices forming paths of length zero. Then, the GPA considers all edges sequentially in order of decreasing weight. If an edge connects two endpoints of two different paths or of an odd length path, the edge is used to connect these paths or close the cycle. After all edges have been considered, an optimal matching is extractet from these paths and cycles using dynamic programming.

Furthermore, the matching algorithm is guided by an edge rating function instead of directly operating on edge weights. A key benefit of the rating function is its ability to incorporate additional local information into the matching process. In particular, the authors use the rating functions expansion$^{*2}(\{u,v\}) = \omega(\{u,v\})^2/c(u)c(v)$ and InnerOuter$(\{u,v\}) = \omega(\{u,v\})/(\text{ Out}(u) + \text{Out}(v) - 2\omega(u,v)\text{ })$, with $\text{Out}(v) := \sum_{x \in N(v)} \omega(\{v,x\})$. Expansion^{*2} makes it less likely for heavy vertices to get matched. Consequently, the coarser graph will a have more even vertex weight distribution, which is beneficial for the FM algorithm. Similarly, inner-outer is designed to prevent high-degree vertices from being matched, which contributes to preserving the structure of regular graphs such as meshes during coarsening.

Once the number of vertices falls below $max(60k, n/(60k))$ the coarsening is stopped, and the coarsest graph is partitioned using a recursive bisection approach.

KaFFPa employs two refinement schemes: quotient graph refinement and k-way local search. The quotient graph is a graph in which each node represents a block of the partition and edges connect pairs of blocks with non-empty boundaries. Quotient graph refinement iterates over the edges of the quotient graph and refines the cut between the corresponding pair of adjacent blocks using so-called 2-way refinement. Two different methods for 2-way refinement are proposed, with one method being a variation of the classical FM algorithm. Instead of selecting vertices alternatingly from both blocks, the vertex with the highest gain is selected in each step. The other method is a novel flow-based refinement based on the max-flow min-cut theorem. This theorem states that the maximum flow in a network is equal to the minimum cut separating the source and sink vertices [44]. In essence, the flow-based refinement constructs a max-flow problem between the two blocks and uses the resulting minimum cut as an improved partition boundary. In contrast to quotient graph refinement, k-way local search operates on all k blocks. The first method of this scheme is a k-way version of the FM algorithm. The second method is a more localized variant of k-way FM called multi-try FM. Instead of initially inserting all boundary vertices into the priority queue, they are first added to a todo list. The algorithm then repeatedly selects a vertex from this list and starts a bounded local k-way search around this vertex and its neighboring boundary vertices. If a vertex from the todo list is picked which was already considered in a localized search in the current round, no search is started from this vertex.

Lastly, KaFFPa introduces F-cycles, which extend the concept of iterated multilevel algorithms. In principle, these algorithms repeat coarsening, initial partitioning, and uncoarsening multiple times. After a graph has been partitioned once, the resulting partition is used as the input partition for the subsequent multilevel iteration. In this case, cut edges of the input are excluded from the matching and therefore will not be contracted. Coars-

ening terminates once no further edges can be contracted. Since no cut edges of the input partition were contracted during coarsening, it remains a valid partition of the coarsest graph. This allows the existing partition to be retained instead of computing a new initial partition. Lastly, the standard uncoarsening and refinement steps are performed without any modifications. F-cycles extend this procedure by recursively repeating the multilevel process during uncoarsening. Whenever a level is reached for the second time, coarsening is started again from that graph until no further contractions are possible, followed by another uncoarsening phase. Since the refinement algorithms employed by KaFFPa guarantee non-decreasing partition quality, every newly computed partition is at least as good as the partition used as input.

Combine Operators. Recombination is one of the two mechanisms used by KaFFPaE to generate new individuals, alongside mutation. Instead of proposing a single combine operator, the authors introduce a general framework for combining two individuals. The framework specifies how offspring are constructed from a pair of parent individuals, but leaves the selection of these parents unspecified. Different combine operators can therefore be obtained by instantiating the framework with different types of parent individuals. This general combine operator framework creates a new individual from a partition and a clustering, which are both used as input for KaFFPa. To this end, the authors modify the coarsening phase of KaFFPa such that edges that are cut edges in either the partition or the clustering are not allowed to be matched. The coarsening process then stops once no further edges can be contracted. Afterwards, the input partition is reused as the initial partitioning on the coarsest graph, which is possible since no cut edges of the partition were contracted during coarsening. The uncoarsening and refinement phases then proceed as normal. Since the refinement algorithm guarantees non-decreasing solution quality, the resulting offspring partition is guaranteed to be at least as good as the input partition.

To maintain diversity within the population, KaFFPaE uses an eviction strategy based on similarity between individuals. More precisely, among all individuals whose cut value is worse than or equal to the offspring, the algorithm evicts the individual most similar to the offspring. Differences between individuals are quantified by the size of the symmetric difference of their respective sets of cut edges.

The authors present three different instantiations of the general combine operator framework. The first one is called *classical combine* and it uses two individuals / partitions from the population, which are picked using tournament selection. The second is called *cross combine* and it combines an individual with a newly generated partition, which is computed using KaFFPa with randomly chosen parameters $k' \in [k/4, 4k]$ and $\epsilon' \in [\epsilon, 4\epsilon]$. The third instantiation, *natural cuts*, combines an individual with a clustering computed using the concept of natural cuts. Natural cuts were introduced by Delling et al. [45] in the context of partitioning of road networks. In essence, they correspond to sparse cuts that separate a local region of the graph from the remaining graph.

Mutation Operators. The second mechanism for creating individuals is mutation. KaFFPaE introduces two mutation operators which perform either a normal or a modified F-cycle on a random individual. Distinguishing the two mutation operators is the

treatment of the initial partition: the normal variant reuses the input partition, whereas the modified variant computes a new initial partition.

Experimental Evaluation. Experiments on the Walshaw benchmark demonstrate the effectiveness of KAFFPAE: the algorithm is able to recompute or improve approximately 76% of the benchmark entries. The authors additionally show that the proposed combine operators consistently outperform simple restarts of KAFFPA.

3.4 Walshaw Benchmark

The Walshaw benchmark archive, created by Soper, Walshaw, and Cross [38], has been in operation since 2000 and is widely regarded as the standard benchmark suite for graph partitioning algorithms. The archive contains 34 real-world graphs originating from various application domains, including finite element simulations, matrix computations, VLSI design, and shortest path problems. In addition to the graph instances themselves, the archive maintains the best known partitions found so far for a range of partitioning parameters. Specifically, partitions are sought for $k \in \{2, 4, 8, 16, 32, 64\}$ and imbalance parameters $\epsilon \in \{0, 0.01, 0.03, 0.05\}$. Running time is not considered in the evaluation. Instead, the primary objective is to obtain ϵ-balanced partitions with the smallest possible edge cut. The archive is publicly accessible and continuously updated. Researchers may submit newly computed partitions, and improved solutions are subsequently listed as the current best results on the Walshaw benchmark website.

CHAPTER 4

MEMHEIPA

Algorithms such as JET [15] have demonstrated that massively parallel architectures can significantly accelerate the partitioning process. However, a relevant research gap remains: While current GPU-based algorithms offer superior speed, they often produce solutions of lower quality compared to the most sophisticated CPU-based alternatives. One of the most successful strategies for achieving high-quality partitions is the use of memetic algorithms. Specifically, Benlic et al. [40] proposed a highly effective multilevel memetic approach that integrates the memetic refinement directly into the multilevel scheme. While this approach is able to recompute or improve many entries in the Walshaw benchmark, it has traditionally required considerable computing time on CPU architectures. Despite the potential of memetic schemes, they have yet to be integrated into high-performance GPU partitioners. Hence, in this chapter, we bridge this gap by introducing MEMHEIPA (Memetic Heidelberg Partitioning) a novel multilevel memetic algorithm designed for the GPU. For this, we adopt the multilevel scheme of JET and extend it with a GPU-adaptation of the memetic components introduced by Benlic et al.

The rest of this chapter is organized as follows: We start with an overview of our algorithm and describe how we operate on individuals of our population in parallel, in Section 4.1. In Section 4.2 we describe how we instantiate the basic components of our multilevel algorithm (coarsening, initial partitioning and uncoarsening). Lastly, Section 4.3 describes our memetic components, including the memetic refinement procedure for generating new individuals, our novel population management strategy, and our mutation operator.

4.1 Algorithm Overview

Our memetic algorithm encodes each individual $I \in POP$ as a k-way partition of the input graph. The fitness of an individual is defined as the edge cut of the corresponding partition,

i.e., $f(I)$ measures the total weight of edges running between different blocks. Algorithm 1 gives an overview of our approach.

We start with coarsening and contracting the graph until it has less than $8 \cdot k$ vertices (Lines 2–5). Then we create a set of partitions of this coarsest graph which becomes our initial population (Line 6). This population is subsequently projected back through the multilevel hierarchy, such that each individual is uncoarsened and refined on each level (Lines 8–11). During the uncoarsening process, we apply our memetic operators at every level to generate new offspring (Line 12). The memetic refinement operator is applied throughout, whereas mutation is only applied on a specified fraction of the final levels (Lines 14–15). Since mutation provides non-decreasing solution quality but is computationally expensive, this design allows us to balance runtime and solution quality by only activating it on a certain percent of levels. Before applying our costly mutation operator, we eliminating less promising individuals and thereby reduce the population size (Line 13). Finally, the best individual in the population is returned as the output of the algorithm.

4.1.1 GPU Parallelization

Population-based metaheuristics are naturally amenable to parallelization, since individuals can be processed independently. On CPUs, this is typically exploited by partitioning the population across multiple threads or cores, where each thread or core operates indepen-

Algorithm 1 MEMHEIPA Overview

Function MEMHEIPA (G_0: *Graph*)

1 $\ell \leftarrow 0$ // ℓ = current level
2 **while** $n_\ell > 8 \cdot k$ **do**
3 $M \leftarrow$ Coarsen(G_ℓ) // M = Mapping
4 $G_{\ell+1} \leftarrow$ Contract(G_ℓ, M)
5 $\ell \leftarrow \ell + 1$
6 $POP_\ell \leftarrow$ ComputeInitialPopulation(G_ℓ)
7 $\ell_{max} \leftarrow \ell$
8 **while** $\ell > 0$ **do**
9 $\ell \leftarrow \ell - 1$
10 $POP_\ell \leftarrow$ UncoarsenPopulation($POP_{\ell+1}$, G_ℓ)
11 $POP_\ell \leftarrow$ RefinePopulation(POP_ℓ)
12 $POP_\ell \leftarrow$ MemeticRefinement(POP_ℓ)
13 $POP_\ell \leftarrow$ ControlSize(POP_ℓ, ℓ)
14 **if** ℓ / ℓ_{max} *is in mutation percentile* **then**
15 $POP_\ell \leftarrow$ Mutation(G_ℓ,POP_ℓ)
16 return arg min$_{I \in POP_0}(f(I))$

dently on its assigned subpopulation. This works well because no communication between the cores is necessary. In our setting, however, computation is offloaded to a single GPU. This makes parallelization more complex, since the GPU represents a shared resource . However GPUs are able to exploit task parallelism and process work issued by different CPU threads. The CPU threads offload their work to the GPU via *kernels*: functions which run on the GPU. The GPU offers different work queues into which CPU threads can commit their kernels. By splitting the population across CPU threads and assigning each a unique work queue, we can efficiently parallelize our memetic algorithm [46, 47]. The main drawback of this approach is its increased GPU memory consumption. Since each CPU thread maintains its own data structures on the GPU, the achievable degree of parallelism is ultimately limited by the available GPU memory

4.2 Multilevel Components

4.2.1 Coarsening And Contraction

We employ a GPU-optimized *two-hop matcher* as proposed in [15]. The main idea is to start with a heavy edge matching and move on to *leaf*, *twin* and *relative* matchings if more than 25 % of vertices stay unmatched. After computing a matching, all matched vertex pairs are contracted to form a smaller graph. Our approach is a slight adaptation of the fine-grained per-vertex hashing scheme proposed in [15]. The main idea to parallelize contraction, is to assign each coarse vertex a dedicated subarray in two hash tables and insert the edges and their weights into these hash tables concurrently using atomic operations. The hash tables are sized using upper bounds, thus a final step is required to extract the actual graph from these arrays.

4.2.2 Initial Partitioning

Once the coarsest graph has been created, we create a set of initial partitions which will form our starting population. Since the size of the coarsest graph is below $8 \cdot k$ vertices, it is too small to exploit enough parallelism on the GPU. For this reason we move the graph to the CPU and use a sequential solver to partition it. In our implementation we make use of KaFFPa in its preconfiguration *strong*.

4.2.3 Uncoarsening

In the uncoarsening step, the coarser graph $G_{\ell+1}$ is replaced by the next finer graph G_ℓ. This can be performed trivially by memorizing the hierarchy of graphs, which is created during coarsening and contracting. What is left, is to project each individuals $I \in POP_{\ell+1}$

to $I' \in POP_\ell$ such that they are a valid partition for the graph G_ℓ. Let M be the mapping which was used to contract G_ℓ into $G_{\ell+1}$. For each individual $I \in POP_{\ell+1}$ we set $I'[v] = I[M[v]], v \in G_\ell$.

4.2.4 Refinement

We refine the population by applying the refinement algorithm proposed in [15] to all individuals of our population. The refinement algorithm alternates between an unconstrained label propagation phase and subsequent rebalancing. The unconstrained label propagation first computes gain values for all vertices and filters promising moves. Then, the afterburner recomputes gains under an approximation of the next partition state. To restore balance, two rebalancing strategies are employed, which move vertices out of overloaded blocks while aiming to minimize the resulting loss in solution quality.

4.3 Memetic Components

4.3.1 Memetic Refinement

To efficiently explore the search landscape and escape local minima, a way to create new solutions and bring diversity into our population is crucial. To this end, we introduce a memetic refinement which creates new solutions using a specialized crossover operator. The procedure is shown in Algorithm 2.

For each offspring we do the following steps: First we select p parents using tournament selection (Line 2), in which λ individuals are randomly choosen and the one with the best fitness value is selected as a parent, unless he is already among the set of parents. Then we produce an offspring using the Backbone Based Crossover (BBC) operator on these parents (Line 3), which will be described below. The new offspring is further refined using JET's refinement (Line 4). Then, if the new offspring is better than the fittest individual or its distance to the population $D_{0,POP}$ is greater than the minimum distance in the population D_{min} (Line 5), we insert it into the population and evict the individual with the worst goodness score (Lines 6–7), with the details on these measures described below.

We measure distances with the partition-distance function $d(\cdot\,,\cdot)$, from [41] which takes as inputs two partitions I_1 and I_2 and returns the number of vertices which one would need to move to transform one of the partitions into the other. Given a population $POP = \{I_1, ..., I_{|POP|}\}$ the value $D_{i,POP} = min\{d(I_i, I_j) : I_j \in POP, i \neq j\}$ gives us the minimum distance between individual I_i and any other individual in the population. The value $D_{min} = min\{D_{i,POP} : i \in 1 \dots |POP|\}$ gives us the minimum distance in the whole population. The goodness score of an individual is given by $g(i) = f(I_i) + \frac{\beta}{D_{i,POP}}$. The hyperparameter β determines how much value is given to diversity. We orient ourselves at [40] and set $\beta = 0.08 * |V|$.

Algorithm 2 Memetic Refinement Overview

Function `MemeticRefinement` (POP_ℓ)

1 **for** *Number Crossovers* θ **do in parallel**
2 $P = \{I_1, \ldots, I_p\} \leftarrow$ TournamentSelection `// P = Parents`
3 $I_0 \leftarrow$ BBC(P) `// I_0 = Offspring`
4 Refinement(I_0)
5 **if** $D_{I_0,POP} > D_{min}$ *or* $f(I_0) < f(I_{best})$ **then**
6 $I_{worst} \leftarrow \arg\max_{I \in POP}(g(I))$
7 $POP_\ell \leftarrow (POP_\ell \cup I_0) \setminus I_{worst}$

4.3.2 Backbone Based Crossover

The crossover operator forms the core of our memetic refinement. Its primary purpose is to generate offspring that are both of high quality and sufficiently different from the existing population. The BBC proposed by Benlic et al. [40] has been shown to achieve this balance between solution quality and diversity. However, since their algorithm was designed for the sequential setting, we now present our parallel version.

The general idea of the BBC is to identify k large sets of vertices which are grouped together in all parent individuals and transmit these sets to the offspring. The remaining vertices are then randomly distributed across the blocks to create diversity (for more detail refer to section 3.3.1). Figure 4.1 shows a small example of the BBC applied to two parent individuals. To parallelize the BBC, three tasks have to be addressed: identifying sets

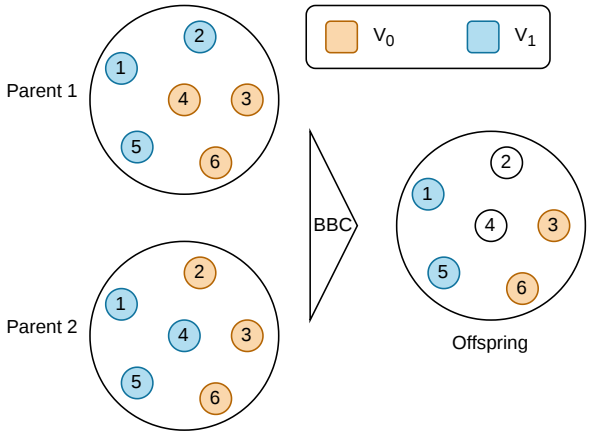


Figure 4.1: Small example of the BBC operator. The vertex set $V = \{1, \ldots, 6\}$ is partitioned into the two sets, highlighted with the two colors blue and orange. The backbone $B = \{\{1, 5\}, \{3, 6\}\}$ is trasmitted to the offspring. The leftover vertices 2 and 4 are highlighted in white. These need to be assigned a color randomly.

of vertices that are grouped together in all parents, selecting which of these sets should be transferred to the offspring, and finally assigning partition IDs to all vertices. The key observation is that such sets can be identified through the block IDs the vertices have among the parent individuals. If two vertices u and v are grouped together in a parent individual I_i they have the same block ID w.r.t. that parent, i.e. $I_i[u] = I_i[v]$. Consequently, if two vertices u and v are grouped together in all parent individuals, they have same block ID among all parents, i.e. $I_1[u] = I_1[v] \wedge I_2[u] = I_2[v] \wedge ... \wedge I_p[u] = I_p[v]$. This means, one grouping can be uniquly identified by a vector of block IDs $< x_1, x_2, ..., x_p >$ where each $x_i \in \{1, ..., k\}$ defines the block ID a vertex has in the parent I_i. Figure 4.2 illustrates this.

Since there are p parents and k different values x_i can take, there are k^p different possible sets of groupings. To decide which of these sets to transmit to the offspring, we make use of a simple greedy heuristic and pick the k biggest sets. We can find the biggest sets by first counting how often each ID vector is present among the set of vertices, and second using the k most frequent ID vectors, since they correspond to the k biggest sets of vertex groupings. These sets will be used to initialize the blocks $V_1, ..., V_k$ of our offspring I_0. In a final step, all vertices that were not assigned during initialization are distributed randomly among the blocks.

The concrete algorithm is shown in Algorithm 3. The algorithm starts by initializing the offspring I_0 as an empty partition (Line 1) and by creating k^p buckets, which are used to count the frequency of each ID vector (Line 2). Counting the frequency can be done in parallel by letting each vertex add itself to a bucket (Lines 3–5). To map an ID vector of a vertex to a bucket index, the tuple of partition IDs is transformed into a single key. One simple way to achieve this is to encode the partition IDs as a bitstring and interpret

Algorithm 3 BackboneBasedCrossover (BBC)

Function `BackboneBasedCrossover` (G_ℓ, P : *Set of Parents*)

```
1  I_0 ← Zeros(n_l)
2  buckets ← Zeros(k^|P|)
3  for v ∈ V do in parallel
4      key ← DetermineKey(v, P)
5      buckets[key]++
6  ParallelSort(buckets)
7  for v ∈ V do in parallel
8      key ← DetermineKey(v, P)
9      for i = 1 to k do
10         if key == buckets[i] then
11             I_0[v] = i
12 AssignLeftoversRandomly
13 return I_0
```

this bitstring as an integer value. The procedure is illustrated in Figure 4.3. This approach works well for small values of p and k. In our setting, p is typically between 2 and 4, and k is at most 1024, such that 64-bit integers are sufficient to represent all keys. For larger values of p, this representation becomes impractical, since the number of possible buckets grows exponentially and the memory consumption increases significantly. After computing the frequency of all ID vectors, the buckets are sorted in descending order (Line 6) to identify the k most frequent vectors, which correspond to the k largest vertex groups. Each vertex can then independently check whether its own ID vector is among these top-k vectors and assign itself to the corresponding block in the offspring partition (Lines 7–11). Finally, all vertices whose ID vectors are not selected among the top-k groups are distributed randomly among the remaining blocks in a separate assignment step.

4.3.3 Population Management

The population management in this work extends the steady-state model proposed by Benlic et al. [40] by introducing a *shrinking* generational model, where the number of individuals shrinks from level to level. The motivation behind this is that refinement is computationally cheap on the contracted graphs and gets more expensive with each projection step. Thus we aim at exploring the search space with many individuals when it is cheap and focus on exploiting a few good individuals when it gets expensive. Furthermore, the limited GPU memory makes it hard to keep a large population throughout the whole memetic algorithm. By shrinking the population size we can avoid spilling individuals to host memory.

The method **ControlSize** in Algorithm 1 is responsible for this shrinking behavior. It first determines how many individuals should survive on the current level and then sequentially eliminates the individuals with the worst goodness score until the desired size is reached. To decide on the population size on a level l we have come up with three different schemes:

1. **Linear.** The first idea is to perform a linear interpolation between the size of the initial population and the desired population size on the final level:

$$|\mathrm{POP}_\ell| = |\mathrm{POP}_0| + \frac{\ell}{\ell_{\max}} \left(|\mathrm{POP}_{\ell_{\max}}| - |\mathrm{POP}_0|\right)$$

2. **Quadratic.** The second idea is to make the populationsize shrink quadratically:

$$|\mathrm{POP}_\ell| = |\mathrm{POP}_0| + \left(\frac{\ell}{\ell_{\max}}\right)^2 \left(|\mathrm{POP}_{\ell_{\max}}| - |\mathrm{POP}_0|\right)$$

3. **Exponential.** The last idea focuses on exponential decay. The population size is cut in half at each level until the desired size is reached. For this we use the following scheme:

$$|\mathrm{POP}_\ell| = \begin{cases} \frac{|\mathrm{POP}_{\ell+1}|}{2}, & \text{if } \frac{|\mathrm{POP}_{\ell+1}|}{2} > |\mathrm{POP}_0| \\ |\mathrm{POP}_0|, & \text{otherwise} \end{cases}$$

The proposed schemes realize different exploration-exploitation trade-offs. In particular, the linear interpolation scheme maintains a larger population over a longer period, whereas the exponential scheme rapidly focuses the search on a small number of individuals. Comparing these variants provides insight into whether the algorithm primarily benefits from early-stage diversity or from prolonged exploitation of a larger population.

4.3.4 Mutation

The crossover operator primarily serves to explore the search space by generating new candidate solutions. However, it does not guarantee that the generated offspring is at least as good as its parents. To strengthen the exploitation of promising solutions, we therefore introduce a mutation operator that guarantees non-decreasing solution quality. Our mutation operator is inspired by KAFFPAE, which employs different *iterated multilevel schemes* as mutation operators [42]. Iterated multilevel schemes, first introduced in [48, 49] and commonly referred to as *V-cycles*, provide an efficient way of refining an existing partition by reusing it within another multilevel cycle.

Algorithm 4 summarizes our mutation operator and provides more details on V-cycles. For each individual I_ℓ, we first sample a uniformly distributed random number $X \sim \mathcal{U}[0, 1]$ (Line 2), and if X is smaller than or equal to the mutation rate (Line 3), a V-cycle is performed on the individual. The V-cycle starts by repeatedly coarsening and contracting the input graph (Lines 5–8). However, during coarsening, cut edges of the input partition I_ℓ are not allowed to be matched. Furthermore, coarsening terminates as soon as no more edges can be contracted. During contraction, we additionally contract the input partition: Whenever two vertices u and v are contracted into a coarse vertex x, the vertex x inherits their common block ID. Note that this is well-defined because cut edges are never contracted, implying that u and v always belong to the same block. Consequently, after the coarsening phase terminates, the contracted partition I_ℓ remains a valid partition of the coarsest graph. During the uncoarsening phase, the partition is projected back to the original graph and refined on each level (Lines 9–12). Since our refinement algorithm guarantees non-decreasing solution quality, the resulting partition is guaranteed to be at least as good as the input partition.

Algorithm 4 Mutation

Function `Mutation(`G_ℓ, POP_ℓ`)`

1 **for** *Individual* I_ℓ *in* POP_ℓ **do in parallel**
2 Let $X \sim \mathcal{U}[0,1]$ `// generate random probability`
3 **if** $X \leq$ *MutationRate* **then**
4 $\ell' \leftarrow \ell$
5 **while** G_ℓ *has contractable edges* **do**
6 $M \leftarrow$ Coarsen(G_ℓ, I_ℓ) `//` M `= Mapping`
7 $G_{\ell+1}, I_{\ell+1} \leftarrow$ Contract(G_ℓ, I_ℓ, M)
8 $\ell \leftarrow \ell + 1$
9 **while** $\ell > \ell'$ **do**
10 $\ell \leftarrow \ell - 1$
11 $I_\ell \leftarrow$ Uncoarsen($I_{\ell+1}$)
12 $I_\ell \leftarrow$ Refine(I_ℓ)

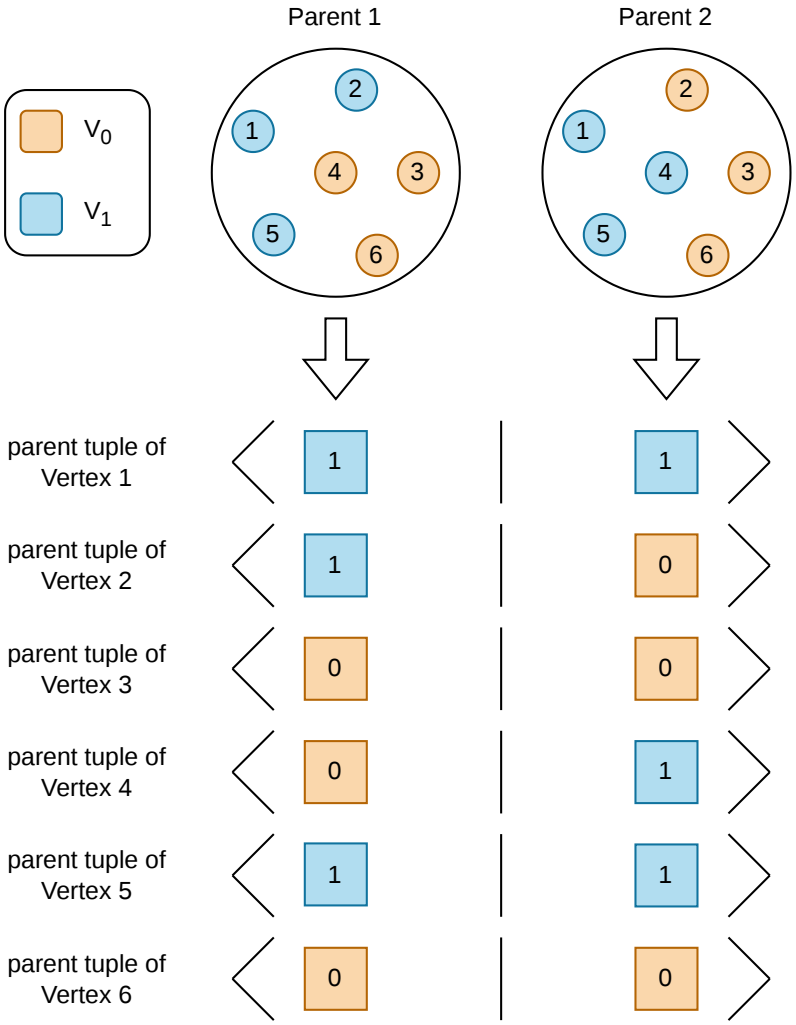


Figure 4.2: Small example of ID vectors / parent tuples. The vertex set $V = \{1, \ldots, 6\}$ is partitioned into two sets, highlighted with the two colors blue and orange. Each vertex is assigned a vector $< x_1, x_2 >$ where x_i denotes the partition ID the vertex has in parent i. Notice that two vertices with the same parent tuples are grouped together in all parents.

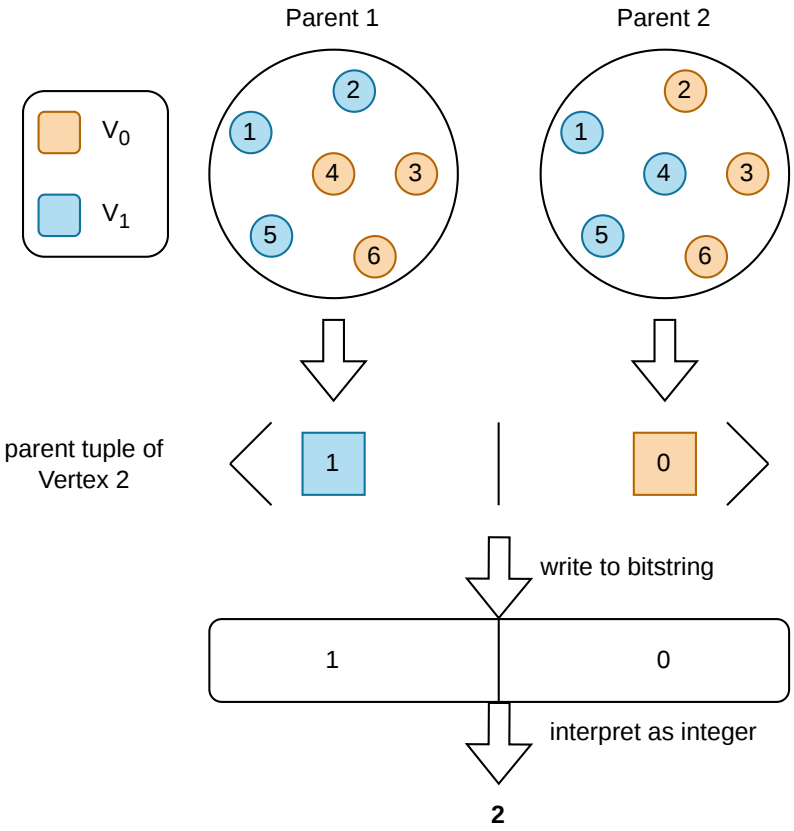


Figure 4.3: Example illustrating the computation of the key for a single vertex. First, the ID vector for vertex two is determined. Next, each number of the ID vector is written into a bit string. The resulting bit string is then interpreted as an integer, which serves as the key.

CHAPTER 5

Exploiting CPU and GPU

Despite the prevalence of CPU-GPU architectures in modern supercomputers, comparatively little attention has been paid to developing graph partitioning algorithms that are able to fully utilize these systems. Most existing graph partitioners are designed either for CPUs or for GPUs and therefore only utilize a subset of the computational resources available in a heterogeneous system. In this chapter, we present a hybrid CPU-GPU graph partitioner which closes this gap. Our novel algorithm is obtained by integrating our GPU-based partitioner into the distributed CPU graph partitioner KaFFPaE.

We start this chapter with a brief overview of KaFFPaE in Section 5.1 and then explain our integration strategy in Section 5.2.

5.1 Overview of KaFFPaE

Before describing our modifications, we briefly summarize the overall structure of KaFFPaE and refer to Section 3.3.2 for a detailed discussion. KaFFPaE is a distributed evolutionary algorithm that is executed on a set of processing elements (PEs). Each PE maintains and evolves its own local population while occasionally exchanging solutions with other PEs. More precisely, each PE executes Algorithm 5 using a different random seed. During the initialization phase (Line 1), the maximum population size S_{max} is determined. Afterwards, the algorithm repeatedly performs rounds until the user-specified time limit is reached (Line 2). Each round consists of a local partitioning phase, which aims at improving the local population (Lines 3–19), followed by a communication phase, which aims at spreading the currently best solutions throughout the set of PEs (Lines 20–25).

The local partitioning phase consists of a fixed number of iterations, where each iteration generates a new individual. If the population has not yet reached its target size S_{max}, a new individual is generated using KaFFPa and inserted into the population (Lines 6–8). Once the population has reached its maximum size, the population is refined using KaFFPaE's mutation and combine operators (Lines 10–19). A mutation selects an individual from the

Algorithm 5 Overview: KaFFPaE (this code is performed by each PE)

Function `KaFFPaE` (G, POP)

Determine Population Size S_{max}
while $t_{now} < t_{max}$ **do**
// **Begin Local Partitioning Phase**
  $iter \leftarrow 0$
  while $iter < iter_{max}$ **do**
    $iter \leftarrow iter + 1$
    if $|POP| < S_{max}$ **then**
      $I_0 \leftarrow$ KaFFPa
      $POP \leftarrow POP \cup I_0$
    else
      Let $X \sim \mathcal{U}(\{0, 1, \ldots, 9\})$
      if $X = 0$ **then**
        $I_0 \leftarrow$ Mutation
      else
        $P_1, P_2 \leftarrow CHOOSE(POP)$
        $I_0 \leftarrow$ Combine(P_1, P_2)
      $Worse \leftarrow \{I \in POP : f(I) \geq f(I_0)\}$
      if $Worse \neq \emptyset$ **then**
        $I_{evict} \leftarrow \arg\max_{I \in Worse}(similarity(I, I_0))$
        $POP \leftarrow \{POP \cup I_0\} \setminus I_{evict}$
// **End Local Partitioning Phase**
// **Begin Communication Phase**
  for $i \in \{1...log(|PEs|)\}$ **do**
    dest $\leftarrow CHOOSE(PEs)$
    $I_{best} \leftarrow \arg\min_{I \in POP}(f(I))$
    Send I_{best} to dest
    receive incoming partitions $\{I_1^{in}, ..., I_x^{in}\}$
    Insert $\{I_1^{in}, ..., I_x^{in}\}$ into POP if possible
// **End Communication Phase**
Collect and return best partition among all PEs

population uniformly at random and performs a F-cycle on it. During a combine, KaFFPa is provided two parent individuals and modified to create an offspring in the following way: Throughout the coarsening phase of KaFFPa the cut edges of both parents are not allowed to be contracted and the coarsening stops once no edges can be contracted any-

more. One of the parent-partitions is then used as the initial partition of the coarsest graph and KAFFPA then performs uncoarsening and refinement as usual. Whether to perform a mutation or recombination is choosen randomly, with a ratio of 1:9 between mutation and combine operations (Lines 10–11). Lastly, the newly generated offspring is inserted into the population, if there exists an individual with a worse edge cut (Lines 16–17). In this case, the individual which is the most similar to the offspring is evicted (Lines 18–19). Similarity between two individuals is measured by the size of the symmetric difference of their corresponding sets of cut edges.

The communication phase is organized in $\log(|PEs|)$ rounds and aims at spreading the currently best solutions throughout the set of PEs. In each communication step, a PE first sends its currently best partition to a randomly selected eligible PE (Lines 21–23). Initially, all other PEs are eligible communication partners. However, after a PE y has sent a message to a PE x, the target PE x becomes temporarily ineligible for y in the current communication phase. After sending a message, a PE incorporates incoming partitions into its local population if they improve the overall population quality (Lines 24–25). Incorporating incoming partitions follows the same rules as for newly generated offspring (Lines 16–19). If a PE receives a partition that improves its current best solution, all other PEs become eligible again.

The algorithm operates fully asynchronously. In particular, during the communication phase, PEs neither wait for acknowledgements nor for incoming messages. Consequently, no global synchronization is required between PEs. Once the time limit is reached, the best partition found among all PEs is collected and returned.

5.2 Integration into KAFFPAE

In this chapter, we describe how the GPU partitioner presented in Chapter 4 is integrated into KAFFPAE. The overall procedure is shown in Algorithm 6. At the beginning of the algorithm, GPUs are assigned to the first PEs. PEs with a GPU perform partitioning using both CPU and GPU resources. The other PEs behave identically to the original version of KAFFPAE. In each round of the algorithm, a PE with GPU support spawns two worker threads: a CPU-worker and a GPU-worker (Lines 8–9). Both workers independently perform partitioning and generate new individuals. The CPU-worker executes the local partition phase as described in the previous section (Algorithm 5, Lines 3–19). The GPU thread is primarily responsible for launching GPU kernels and performs only minimal host-side computation. Therefore, it introduces negligible CPU overhead, allowing both threads to run efficiently in parallel. After the workers finish, their results are merged (Line 13) and the communication phase (line 16) proceeds as in the original version of KAFFPAE (Algorithm 5, Lines 20–25).

5.2.1 Initialization and Merging of Temporary Populations

In KAFFPAE, each PE maintains a local population that is refined throughout the execution of the algorithm. Allowing both worker threads to directly operate on this population would require extensive synchronization and would substantially reduce parallelism. To avoid this issue, each worker thread is assigned its own population and can therefore operate independently of the other thread. The CPU-thread operates on the population of the PE and the GPU-thread operates on its own population (Algorithm 6, Lines 8–9).

We investigate two different strategies for initializing the GPU-thread's population. The most straightforward approach, which we refer to as the *Copy-Model*, initializes the GPU-workers population as a copy of the PE's population. As an alternative, we consider a *Producer-Consumer-Model*. Here, the GPU-worker is primarily responsible for introducing diversity by generating new solutions, while the CPU-worker focuses on refining existing ones. Consequently, the GPU-worker starts with an empty population and gradually builds its own set of individuals, which will be described in more detail in Chapter 5.2.2.

Since communication in KAFFPAE takes place on the level of PEs, the two populations maintained by the worker threads must eventually be combined into a single population of the PE. We therefore merge the two populations before the communication step (Algo-

Algorithm 6 Overview: KAFFPAE with GPU (this code is performed by each PE)

```
Function KaFFPaE+GPU(G, POP)
1   Estimate Population Size S_max
2   while t_now < t_max do
3       if hasGPU then
4           if Producer-Consumer-Model then
5               POP_GPU ← ∅
6           else
7               POP_GPU ← copy(POP)
8           spawn CPU-Worker(G, POP) // Local Partitioning Phase
9           spawn GPU-Worker(G, POP_GPU)
10          Wait for CPU-Worker
11          signal GPU-Worker to stop
12          Wait for GPU-Worker
13          POP ← Merge(POP, POP_GPU)
14      else
15          // Local Partitioning Phase
16      // Communication Phase
17  Collect best partition among all PEs
```

rithm 6, Line 13), according to solution quality and repeatedly select the best individuals until the target population size is reached.

A naive merge strategy can, however, significantly reduce diversity. This issue is particularly pronounced in the Copy-Model, where both worker populations initially contain the same individuals. If a high-quality individual survives in both populations, it may be inserted multiple times into the merged population. Repeating this process over several rounds can cause a small number of individuals to dominate the population. Since preserving diversity is of high importance to the effectiveness of a memetic algorithm, we employ a simple diversity-preserving heuristic during merging. Whenever an individual is considered for insertion, we check whether the merged population already contains an individual with the same edge cut. If this is the case, the individual is discarded. While this heuristic may reject structurally different partitions with identical objective values, it provides a computationally inexpensive approximation that avoids the significantly higher cost of comparing partitions directly.

5.2.2 GPU-Worker

Algorithm 7 shows how the GPU-worker thread refines the population using our newly presented partitioner from Chapter 4. The algorithm operates in an iterative fashion, where in each iteration a new individual is generated. If the current population has not yet reached its maximum size S_{max}, a new individual is created using MEMHEIPA (Lines 4–6). Otherwise, the GPU-worker samples a variable X uniformly at random from the Interval $[0, 1]$ (Line 8) and uses it to select between two modes. With probability 0.4, it tries to increase the diversity of the population (Lines 9–10), and with probability 0.6, it tries to improve an existing solution (Lines 11–13). To increase diversity, a new individual is generated by applying MEMHEIPA in its standard form. To improve an existing solution, an individual is selected uniformly at random from the current population (Line 12) and used as input to MEMHEIPA. In this case, MEMHEIPA performs a V-cycle: During coarsening, edges that are cut edges in the input partition are excluded from matching and contraction. Once no edges can be contracted anymore, the initial population is generated such that one individual directly reflects the input partition, while all remaining individuals are created using the initial partitioning algorithm. Refinement is then performed as usual. The newly generated individual replaces the worst individual in the population if it has a better edge cut (Lines 14–16). The algorithm repeats these iterations until either a maximum number of iterations is reached or the CPU-worker thread has finished (Line 2). This stopping criterion is necessary because CPU and GPU partitioning have different and hard-to-predict runtimes. The signaling mechanism ensures that both worker threads terminate within a comparable time frame.

Algorithm 7 Overview: Local Partitioning With the GPU

Function `GPU-Worker(`G, POP`)`

```
1  iter ← 0
2  while not CPU thread finished AND iter < iter_max do
3      iter ← iter + 1
4      if |POP| < S_max then
5          I_0 ← MemHeiPa
6          POP ← POP ∪ I_0
7      else
8          Let X ∼ U(0, 1)
9          if X < 0.4 then
10             I_0 ← MemHeiPa
11         else
12             P ← CHOOSE(POP)
13             I_0 ← MemHeiPa (P)
14         I_worst ← arg max_{I∈POP} f(I)
15         if f(I_0) < f(I_worst) then
16             POP ← {POP ∪ I_0} \ I_worst
```

CHAPTER 6

Experimental Evaluation

6.1 Methodology

Hardware. We perform our experiments on two different machines. *Machine A* is used for CPU algorithms and *Machine B* is used for algorithms which require a GPU.

Machine A: The machine is equipped with two AMD EPYC 9454 processors (96 cores with a total of 192 threads, 2.75 GHz base frequency with a max boost clock of 3.8GHz) and 384 GiB of RAM. The operating system used is Red Hat Enterprise Linux 9.4.

Machine B: The machine also uses two AMD EPYC 9454, but offers 768 GiB of RAM and 4 NVIDIA H100 GPUs (with 94 GiB of VRAM each). The operating system used is Red Hat Enterprise Linux (RHEL) 9.4.

Software. Our program is written in C++20 and Kokkos 5.0.0. It is compiled using g++ (GCC) 14.2.0 and nvcc (CUDA toolkit) 12.8 for the GPU code, with full optimization enabled (-O3). For KAFFPAE we use OpenMPI 5.0.7. and run the executable with 16 MPI ranks. We compare against JET [15] and G-KWAY [33] as GPU baselines, and against METIS [8] and KAFFPA [10] as CPU baselines. For the individual comparisons, we use JET with its preconfigurations *default* and *ultra* , METIS version 5.1.0 with the default configuration, and KAFFPA version 3.25 with the *strong* preconfiguration.

Performance Profiles. To compare different algorithms or algorithm configurations, we make use of performance profiles [50]. We use them to compare the solution quality, measured by the achieved edge cut. The x-axis is labeled with the performance ratio $\tau \geq 1$, which describes by what factor the solution quality of an algorithm differs from the best solution quality found among all compared methods. The y-axis indicates the fraction of instances. A point (τ, p) indicates that on a fraction p of all instances, the algorithm achieves a solution whose edge cut is at most a factor τ larger than the best solution among all compared algorithms.

More formally, let $\mathcal{A}$ denote the set of all algorithms, $\mathcal{I}$ the set of all benchmark instances, $f_A(I)$ the edge cut of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$ and $\text{Best}(I) = min_{A \in \mathcal{A}} f_A(I)$ the best solution on instance I. The performance profile $\rho_A : \mathbb{R}_{\geq 1} \mapsto [0, 1]$ for an algorithm A is defined as

$$\rho_A(\tau) := \frac{|\{I \in \mathcal{I} : f_A(I) \leq \tau \cdot \text{Best}(I)\}|}{|\mathcal{I}|}$$

and describes the fraction of instances for which an algorithm A achieves an edge cut which is within a factor τ of the best edge cut. In general, large fractions at small values of τ indicate good performance. In the following, the term performance profile refers both to the function $\rho_A(\tau)$ of a single algorithm and to the resulting plot containing the performance profiles of all algorithms in $\mathcal{A}$.

Wilcoxon Signed-Rank Test. Additionally, we make use of the Wilcoxon signed-rank test [51], to assess whether the differences between two algorithms or algorithmic configurations are statistically significant.

If one algorithm truly performs better than the other, we would expect this algorithm to achieve smaller edge cut values on more instances than the other. Hence, if we consider the differences in the edge cut values, we expect them to systematically deviate from zero, with more and/or larger differences favoring one algorithm. Likewise, if both algorithms perform similarly, we would expect the differences to be centered around zero, such that positive and negative differnces occur in a balanced way.

The Wilcoxon signed-rank test formalizes this idea by considering both the sign and the magnitude of the differences. Instead of using the raw values directly, it ranks them in absolute value, which reduces sensitivity to extreme outliers. Based on these signed ranks, the test computes a p-value that quantifies the likelihood that the observed performance difference could have arisen by chance. The smaller the p-value, the less likely it is, that the algorithms coincidentally perform different.

More precisely, the calculation of the test works as follows: Given the edge cuts found by two algorithms $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$ obtained on the same instances, we compute the differences $d_i = a_i - b_i$ and form the set $D = \{d_1, \ldots, d_n\}$. All zero differences are discarded. The remaining values are ranked according to the absolute values $|d_i|$, where the smallest absolute difference receives rank 1, the second smallest rank 2, and so on (ties are assigned average ranks). Each rank is then assigned the sign of its corresponding difference.

We then compute the sums of positive and negative ranks:

$$T^+ = \sum_{d_i > 0} r_i, \quad T^- = \sum_{d_i < 0} r_i.$$

The test statistic is defined as:

$$W = \min(T^+, T^-).$$

Depending on the number of instances, two different approaches are used to evaluate the significance of W. For small sample sizes (typically ≤ 50), W is compared to so called *critical values* W_{crit} to determine statistical significance. The critical values can be obtained from precomputed tables. For larger sample sizes (typically > 50), the test statistic is assumed to be normally distributed. In this case, W's expected value μ_W and standard deviation σ_W are computed. Then, the z-value can be computed using the formula $z = (W - \mu_W)/\sigma_W$, which can be used to calculate the p-value. Finally, the resulting p-value is used to determine statistical significance. We consider $p \leq 0.05$ as statistically significant, meaning that the observed differences are unlikely to have occurred by chance.

Convergence Plots. We use convergence plots to visualize how the solution quality of KAFFPAE evolves over time. Convergence plots show the time on the x-axis and the solution quality on the y-axis. First we start by describing how we create a convergence plot for a single execution of KAFFPAE using P PEs. Before the communication step in each round of the KAFFPAE algorithm, we let a PE document a pair (t, *cut*) where t refers to how long the algorithm has been run and *cut* refers to the best edge cut among the population of the PE. After the algorithm finishes, we are left with P sequences of pairs. We merge these sequences into a single sequence based on the timestamp t, i.e. the final sequence is sorted with respect to t. We refer to this sequence as T^I. Using T^I we compute another sequence T^I_{min} which describes the evolution of the solution quality. We iterate over the entries of T^I and for an entry (t, *cut*) in T^I, we insert the entry (t,$\min_{t' \leq t}(t')$) into T^I_{min}, where $\min_{t' \leq t}(t')$ refers to the smallest edge cut found up to the point t.

Next, we describe how we obtain a convergence sequence for multiple runs of the algorithm. For this, we use the r sequences $T^{I_1}_{min}, \ldots, T^{I_r}_{min}$ we obtained from the r executions of the algorithm. To aggregate these sequences, we first label each entry $(t,$ *cut*$)$ of a sequence $T^{I_i}_{min}$ with the run ID i. This transforms the entries into triples $(t,$ *cut*$, i)$. We then merge all r sequences into a single sequence S based on the timestamp t. The sequence S will thus be sorted with respect to t. From S we derive a new sequence S_A describing the average convergence behavior. To compute this sequence, we maintain a set $C = \{c_1, \ldots, c_r\}$ where c_i denotes the most recently observed cut value of run i. Initially, c_i is set to the first cut value of $T^{I_i}_{min}$. The first entry of S_A is then obtained by computing the arithmetic mean of the values in C. We subsequently process the entries of S in chronological order. Whenever an entry $(t, \textit{cut}, i)$ is encountered, the value c_i is replaced by *cut*. The arithmetic mean $\bar{c}$ of the values in C is then recomputed and the pair $(t, \bar{c})$ is appended to S_A. Consequently, each entry of S_A represents the average of the best solution qualities known for the individual runs at that point in time.

To aggregate runsequences of different graphs or different values for k, we proceed identically, but replace the arithmetic mean with the geometric mean, since the cut values may deviate significantly from each other.

Figure 6.1 illustrates how we average over two different runs (left hand side) and two different graph instances (right hand side) respectively. In the example for averaging over

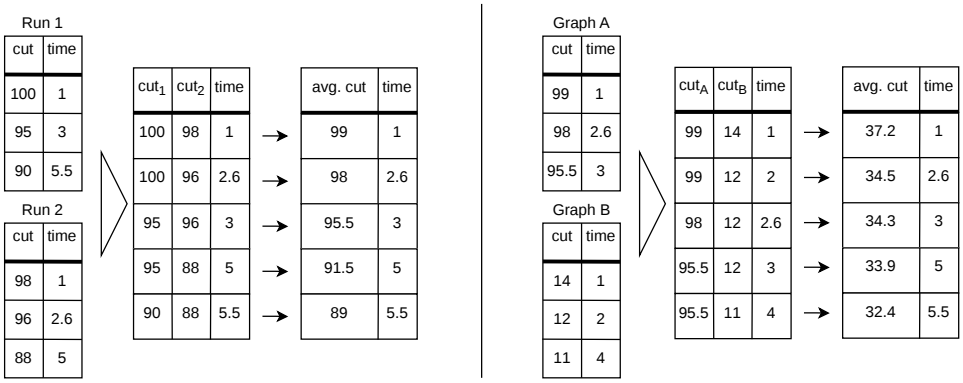


Figure 6.1: A small example showing how an average runsequence is determined from two executions of the algorithm (left) and from two different graph instances (right).

different runs, the two tables on the left represent the convergence histories of two independent executions of the algorithm, i.e. $T_{min}^{I_1}$ and $T_{min}^{I_2}$. The sequence shown in the middle depicts the evolution of the set C, while the averaged convergence sequence S_A is shown on the right. Initially, at time $t = 1$, the set C contains the first cut values of both runs, i.e. $C = \{100, 98\}$. The arithmetic mean is therefore 99, yielding the first entry $(1, 99)$ of S_A. At time $t = 2.6$, the cut value of run 2 improves from 98 to 96. We therefore update the set to $C = \{100, 96\}$, recompute the average, and append the pair $(2.6, 98)$ to S_A. Next, at time $t = 3$, the cut value of run 1 improves to 95. Consequently, C becomes $\{95, 96\}$, resulting in the new entry $(3, 95.5)$ in the averaged convergence sequence. The remaining entries of S_A and the example which averages over two different instances are left out for brevity. The runsequences S_G depicted in the convergence plot is obtained by averaging over all runs, k values and graphs.

6.2 Instances

We use three different datasets throughout our experimental evaluation. The first dataset is used for our tuning experiment and it is depicted in Table 6.1. These graphs are taken from the 10th DIMACS Implementation Challenge on graph partitioning and clustering, which is a widely used benchmark suite [52]. The tuning set covers graphs from different application domains and a wide range of sizes, ensuring that the resulting parameter configuration generalizes across different problem instances.

The second dataset is used for the comparison with JET and is shown in Table 6.2. To ensure a fair comparison, we use the same benchmark set that was employed in the evaluation of JET [53]. The dataset consists of 55 graphs from the SuiteSparse Matrix Collection [54], three graphs from the Open Graph Benchmark [55], four graphs from the Laboratory for Web Algorithmics [56, 57], and one graph from the Walshaw benchmark [38]. In addition,

two synthetic graphs were generated by the authors of JET.

The third dataset is used to compare the original version of KAFFPAE with the hybrid CPU-GPU variant proposed in Chapter 5. Since the original KAFFPAE paper evaluates the algorithm on the Walshaw benchmark [38], we use the same benchmark suite to facilitate a direct comparison. The corresponding set of graphs is shown in Table 6.3. A detailled explanation of the Walshaw benchmark is given in Section 3.4.

The benchmark sets contain many real-world instances from a wide range of application domains. For example, graphs named **kmer_...** originate from bioinformatics applications and represent protein k-mers. Graphs named **fe_...** correspond to computational meshes arising from finite element simulations, and instances such as **europe_osm** represent road networks. In addition to these real-world instances, the benchmark sets also contain several families of generated graphs. Graphs named **rgg_n_2_X** are random geometric graphs in two-dimensional space with 2^X vertices. The vertices correspond to points sampled uniformly from the unit square, and two vertices are connected by an edge if their Euclidean distance is smaller than $0.55\sqrt{\ln(n)/n}$. Graphs named **delaunay_nX** are also based on 2^X randomly generated points in the unit square. However, the delaunay triangulation of the point set determines which vertices to connect by an edge. Finally, graphs named **kron_...** are Kronecker graphs [58], a class of synthetic graphs designed to mimic structural properties commonly observed in real-world networks. They are generated by recursively applying a matrix operation called the *Kronecker product* to an initiator matrix.

Table 6.1: Tuning Graphs

Graph	$\|V\|$	Type
rgg_n_2_16	65 536	Random geometric graph
luxembourg	114 599	Road network
delaunay_n17	131 072	Delaunay triangulation
coauthorsciteseer	227 320	Co-authorship network
af_shell9	504 855	Sparse matrix
ldoor	952 203	Sparse matrix
packing-500x100x100-b050	2 145 852	Numerical simulation mesh
m6	3 501 776	Numerical simulation mesh

Table 6.2: Benchmark graphs used to evaluate JET

Graph	n	m	Graph	n	m
vsp_bcsstk30_500sep_10in_1Kout	58 348	2 016 578	Queen_4147	4 147 110	162 676 087
vsp_vibrobox_scagr7-2c_rlfddd	77 328	435 586	rgg_n_2_22_s0	4 194 299	30 359 197
mycielskian17	98 303	50 122 871	vas_stokes_4M	4 344 906	97 708 521
fe_rotor	99 617	662 431	channel-500x100x100-b050	4 802 000	42 681 372
mycielskian18	196 607	150 466 916	soc-LiveJournal1	4 843 953	42 845 684
dblp-2010	226 413	716 460	cage15	5 154 859	47 022 346
ppa	576 039	21 231 776	ljournal-2008	5 363 186	49 514 271
amazon-2008	735 323	3 523 472	circuit5M	5 555 791	26 981 620
kron_g500-logn20	795 153	44 619 358	enwiki-2021	6 253 852	136 494 815
audikw_1	943 695	38 354 076	indochina-2004	7 320 539	149 054 854
hollywood-2009	1 069 126	56 306 653	grid_3	8 000 000	15 994 000
dielFilterV3real	1 102 824	44 101 598	cube_2	8 000 000	23 880 000
Serena	1 391 349	31 570 176	nlpkkt160	8 345 600	110 586 256
Geo_1438	1 437 960	30 859 365	rgg_n_2_23_s0	8 388 601	63 501 390
Long_Coup_dt0	1 470 152	42 809 420	delaunay_n23	8 388 608	25 165 784
Long_Coup_dt6	1 470 152	42 809 420	wb-edu	8 863 287	44 185 251
Hook_1498	1 498 023	29 709 711	stokes	11 303 355	257 981 313
ML_Geer	1 504 002	54 687 985	nlpkkt200	16 240 000	215 992 816
af_shell10	1 508 065	25 582 130	rgg_n_2_24_s0	16 777 215	132 557 200
kron_g500-logn21	1 543 901	91 040 839	delaunay_n24	16 777 216	50 331 601
Flan_1565	1 564 794	57 920 625	hugebubbles-00000	18 318 143	27 470 081
soc-Pokec	1 632 803	22 301 964	uk-2002	18 459 128	261 556 721
hollywood-2011	1 917 070	114 281 101	hugebubbles-00010	19 458 087	29 179 764
HV15R	2 017 169	162 357 569	hugebubbles-00020	21 198 119	31 790 179
vas_stokes_2M	2 146 677	48 351 779	arabic-2005	22 634 275	552 231 867
Cube_Coup_dt0	2 164 760	62 520 692	road_usa	23 947 347	28 854 312
Cube_Coup_dt6	2 164 760	62 520 692	europe_osm	50 912 018	54 054 660
products	2 385 902	61 806 303	kmer_V2a	53 500 237	57 076 126
Bump_2911	2 852 430	62 409 240	kmer_U1a	64 678 340	66 393 629
citation	2 915 301	30 344 439	kmer_P1a	138 896 082	148 465 346
com-Orkut	3 072 441	117 185 083	kmer_A2a	170 372 459	179 941 739
nlpkkt120	3 542 400	46 651 696	kmer_V1r	214 004 392	232 704 832
com-LiveJournal	3 997 962	34 681 189			

Table 6.3: Walshaw Benchmark Graphs

Graph	n	m	Graph	n	m
add20	2 395	7 462	bcsstk30	28 924	1 007 284
data	2 851	15 093	bcsstk31	35 588	572 914
3elt	4 720	13 722	fe_pwt	36 519	144 794
uk	4 824	6 837	bcsstk32	44 609	985 046
add32	4 960	9 462	fe_body	45 087	163 734
bcsstk33	8 738	291 583	t60k	60 005	89 440
whitaker3	9 800	28 989	wing	62 032	121 544
crack	10 240	30 380	brack2	62 631	366 559
wing_nodal	10 937	75 488	finan512	74 752	261 120
fe_4elt2	11 143	32 818	fe_tooth	78 136	452 591
vibrobox	12 328	165 250	fe_rotor	99 617	662 431
bcsstk29	13 992	302 748	598a	110 971	741 934
4elt	15 606	45 878	fe_ocean	143 437	409 593
fe_sphere	16 386	49 152	144	144 649	1 074 393
cti	16 840	48 232	wave	156 317	1 059 331
memplus	17 758	54 196	m14b	214 765	1 679 018
cs4	22 499	43 858	auto	448 695	3 314 611

6.3 Tuning Experiments

Our memetic algorithm has several hyperparameters that influence its performance. Before comparing our approach to other algorithms, we perform a tuning study to determine a high-quality parameter configuration.

We investigate the following parameters:

- Starting population size $|POP_{l_{max}}|$: Determines the amount of initial partitions of the coarsest graph.
- Final population size $|POP_0|$: Determines how many individuals remain at the end of the algorithm.
- Number of crossovers θ: Determines how many offspring are created on each level of the multilevel algorithm.
- Number of parents $p \in [2, 4]$: Determines how many parents are used to create an offspring.
- Tournament size $\lambda \in [2, |POP_{l_{max}}|]$: Determines how many individuals are compared against each other during tournament selection.
- Type of shrinking function $s \in \{linear, quadratic, exp\}$: Determines how fast the size of the population shrinks during the uncoarsening process.
- Mutation percentile $M_{\%} \in [0, 1]$: Determines on what percentile of final levels the mutation operator is used ($M_{\%} = 0.0 \rightarrow$ no mutation at all, $M_{\%} = 1.0 \rightarrow$ mutation performed on all levels).
- Mutation rate $M_r \in [0, 1]$: Determines the likelyhood of mutating a random individual ($M_r = 0.0 \rightarrow 0\%$ chance, $M_r = 1.0 \rightarrow 100\%$ chance).

Table 6.4 shows the baseline configuration used as a starting point for all tuning experiments. This configuration was chosen because it provided good performance in preliminary experiments during development. Furthermore, we start our experiments with a linear shrinking function and mutation turned off, in order to investigate the effects of different shrinking functions and the mutation operator during our tuning experiments.

For each configuration, we run experiments with $k \in \{8, 16, 32\}$ and a fixed imbalance of $\epsilon = 3\%$. Each experiment is repeated five times using different random seeds. We use four CPU threads for all runs.

The methodology of our tuning experiments is inspired by the coordinate descent algorithm [59]. Coordinate descent optimizes one parameter at a time while keeping all remaining parameters fixed. After tuning a parameter, the best configuration found is used as the

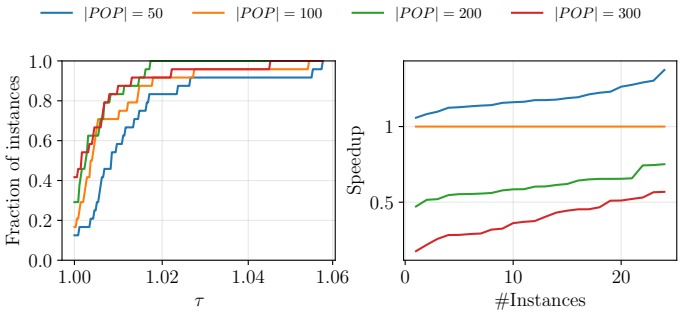
Table 6.4: Starting Configuration of the Algorithm

Parameter	Value
Starting Pop. Size $\|POP_{l_{max}}\|$	100
Final Pop. Size $\|POP_0\|$	10
Num. Crossovers θ	10
Num. Parents p	2
Tournament Size λ	2
Shrinking function s	linear
Mutation Percentile $M_{\%}$	0.0
Mutation Rate M_r	0.0

baseline for subsequent experiments. To compare configurations, we use performance profiles, speedup plots, and tables. In the tables, the column Δ *cut* shows the improvement in the geometric mean edge cut and the column *speedup* shows the geometric mean speedup over all runs.

6.3.1 Starting Population Size

The first parameter we tune is the size of the initial population. Our baseline configuration uses 100 individuals, and we evaluate both smaller and larger population sizes. Figure 6.2 shows the results of our experiments. Overall, we observe a tendency that larger populations improve solution quality. This is reflected in the performance profile, which shows a clear trend: larger population sizes yield better solution quality, with 300 individuals finding the best solution on most instances (42%), followed by 200, 100, and 50. Furthermore, the smaller population size of 50 leads to a decrease of 0.5% in the geometric mean edge cut, whereas the larger populations of 200 and 300 individuals improve the geometric mean edge cut by 0.3% and 0.2%, respectively. These effects may be due to the fact that larger populations allow the algorithm to explore a broader search space and therefore to find better solutions. However, this improvement comes at a significant computational cost: using 300 individuals results in a speedup of 0.43x relative to the baseline. This is mainly due to the increased number of individuals that must be uncoarsened and refined at each level of the multilevel hierarchy. Interestingly, the configuration with 200 individuals achieves a slightly higher average edge cut improvement than 300, despite the latter performing better in terms of best-solution frequency. However, according to the Wilcoxon signed-rank test, this difference is not statistically significant ($p = 0.37$). Considering the trade-off between solution quality and runtime, we select a starting population size of 200 for our final configuration, as it achieves comparable solution quality to 300 while offering a faster runtime.



$\lvert POP_{l_{max}} \rvert$	Δ cut	speedup
50	-0.5%	1.29x
100	0.0%	1.00x
200	0.3%	0.58x
300	0.2%	0.43x

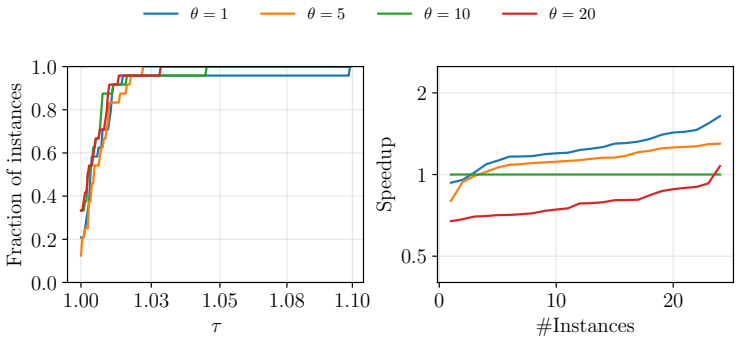
Figure 6.2: Comparison of different starting population sizes. On the left is a performance profile, in the middle a speedup plot and the table on the right shows the improvement of the geometric mean edge cut (Δ cut) and geometric mean speedup (speedup).

6.3.2 Final Population Size

We now investigate the final population size. Our baseline configuration uses a final population size of 10, and we compared this to the values one, three, five and 20. Our experiments indicate that the final population size has no measurable impact on solution quality. The average edge cut improvements range from -0.166% to 0.103%, and none of these differences are statistically significant according to the Wilcoxon signed-rank test. Interestingly, this behavior differs from the results of the previous experiment, where increasing the starting population size led to noticeable quality improvements. This suggests that solution quality benefits more from exploring the search space with many individuals, rather than maintaining a larger set of high-quality solutions. In contrast, runtime is affected by the final population size. Smaller values consistently lead to faster execution times, with the best speedup of $1.10\times$ observed for $|POP_0| = 3$ (with $p = 0.00178$). We therefore use $|POP_0| = 3$ for the remaining experiments.

6.3.3 Number of Crossovers

We proceed with the number of crossovers θ performed on each level of the multilevel algorithm. The baseline is $\theta = 10$ and we explore a range of alternative values. Figure 6.3 shows the results. Regarding solution quality, we observe a slight positive trend: increasing the number of crossovers tends to improve the solution quality marginally. This is visible in the performance profile, where the best results are achieved for $\theta = 10$ and $\theta = 20$. Similarly, the table shows that reducing the number of crossovers leads to a slight decrease in average solution quality. These findings further support the observation that the algorithm benefits especially from the exploration of the search space, since the crossover operator generates new individuals and thereby explores new regions of the solution landscape. However, generating offspring is computationally expensive, as our refinement algorithm is applied to each new individual directly after its creation. This is reflected in the speedup values: using only 1 or 5 crossovers per level results in average speedups of 1.24x and



θ	Δ cut	speedup
1	-0.316%	1.24x
5	-0.100%	1.13x
10	0.000%	1.00x
20	0.060%	0.79x

Figure 6.3: Comparison of different number of crossovers θ.

1.13x, respectively, whereas increasing the number of crossovers to 20 reduces the speedup to 0.79x. We therefore keep the initial configuration of $\theta = 10$, since it provides a similar solution quality to $\theta = 20$ while maintaining a better runtime performance.

6.3.4 Number of Parents

Subsequently, we investigate the number of parents used during each crossover. Our baseline configuration uses two parents, and we additionally evaluate configurations with three and four parents. Larger values could not be considered, since the memory requirements of the crossover operator grow exponentially with the number of parents and exceeded the available GPU memory. Overall, increasing the number of parents does not improve solution quality. Compared to the baseline, using three and four parents changes the geometric mean edge cut by only -0.066% and -0.031%, respectively. Similarly, the impact on runtime is tiny, resulting in average speedups of $1.02\times$ and $1.00\times$ for three and four parents respectively. Since increasing the number of parents provides no improvement but increases the memory consumption, we keep the baseline value of two parents.

6.3.5 Tournament Size

We proceed with investigating the tournament size λ which is relevant for choosing parents. Tournament selection samples λ individuals uniformly at random and selects the individual with the best fitness as a parent. Our baseline uses $\lambda = 2$ and we evaluate a set of larger values. The results are shown in Figure 6.4. It can be seen that all larger tournament sizes outperform the baseline with respect to solution quality. A likely explanation is that larger tournaments increase the probability of selecting high-quality parent individuals, which in turn leads to better offspring. In our experiments, we investigated values up to 100 but did not notice any further improvements after 40. Furthermore, increasing the tournament size also leads to faster execution times. One reason is that the crossover is performed less with increasing values of λ, since it can only be performed when there are

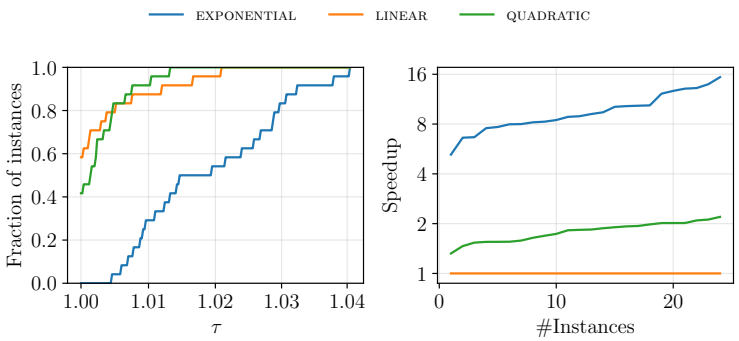
λ	2	10	20	30	40
Δ cut	0.000%	0.351%	0.687%	0.717%	0.954%
speedup	1.00x	0.98x	1.13x	1.09x	1.23x

Figure 6.4: Comparison of different tournament sizes λ.

enough individuals in the population to perform tournament selection. The point where the population size is smaller than the tournament size comes earlier as we increase λ, thus leading to less computational time. Another possible explanation is that the refinement algorithm requires less time to improve the higher-quality offspring. Overall, we select $\lambda = 40$ for the remaining experiments, since it achieved the best improvement in solution quality and execution time.

6.3.6 Shrinking Function

We then compared different shrinking functions, which control how quickly the population size decreases throughout the uncoarsening phase. As a baseline, we perform a linear interpolation between the starting and final population size and compare to a quadratic interpolation and an exponential decay. The results are shown in Figure 6.5. The performance profile shows a clear ranking: the linear shrinking function finds the best solution on 60% of the instances, followed by the quadratic function with 40%, whereas the exponential decay does not achieve the best solution on any instance. This observation is consistent with our previous findings, that utilizing more individuals improves the solution quality. Concerning the speedup, both quadratic and exponential achieve promising average speedups of $1.79\times$ and $9.38\times$ respectively. This is expected, since a smaller population requires less work. Given these results, we choose a quadratic function since it achieves comparable results to linear while being on average $1.79\times$ faster.



s	Δ cut	speedup
exp.	-1.621%	9.38x
linear	0.000%	1.00x
quadratic	0.031%	1.79x

Figure 6.5: Comparison of different shrinking functions.

6.3.7 Mutation

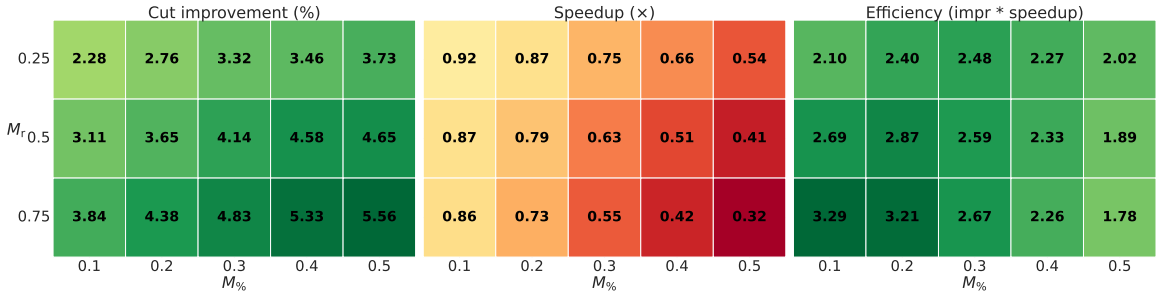


Figure 6.6: Comparison of different mutation percentiles and mutation rates.

Next, we investigate the parameters of our mutation operator. Since the mutation rate M_r and the mutation percentile $M_\%$ interact, we tune both parameters in conjunction. Mutation is applied by iterating over all individuals and performing a V-cycle with a probability determined by the mutation rate. For each individual, a random value in $[0, 1]$ is generated and a V-cycle is performed if this value is smaller than or equal to the mutation rate. The mutation percentile determines the levels of the multilevel hierarchy at which mutation is applied. Small values restrict mutation to the final uncoarsening levels, where the graph is already close to its original size, whereas large values allow mutation to be already performed soon after the initial partitioning phase. Figure 6.6 shows heatmaps of the average edge cut improvement (left), the average speedup (middle), and the efficiency (right) for the evaluated parameter combinations. We define the efficiency as the product of the average edge cut improvement and the average speedup, since this measure favors parameter configurations achieving both high solution quality improvements and high speedups.

Regarding solution quality, we observe that the mutation operator provides a substantial improvement. Both increasing the mutation rate and increasing the mutation percentile lead to better solution quality. However, our experiments showed that increasing the mutation percentile beyond 0.5 yields only marginal additional improvements. Therefore, larger values are omitted from the figure. These improvements come at the cost of increased runtime. Both higher mutation rates and larger mutation percentiles increase the number of mutation operations that are performed and therefore the overall computational effort. Interestingly, applying mutations primarily during the final stages of the multilevel algorithm appears to be considerably more efficient than distributing them more uniformly throughout the hierarchy. This can be seen in the heatmap on the right, where the highest efficiency is achieved by combining a high mutation rate with a small mutation percentile. A possible explanation is that towards the end of the multilevel hierarchy only a small number of high-quality individuals remain. Investing additional refinement effort into these individuals is likely more beneficial than mutating individuals at earlier stages that may later be discarded. We therefore use a mutation rate of 0.75 and a mutation percentile of 0.1 for the remaining experiments, as this configuration achieves the highest efficiency among all

evaluated parameter combinations.

Final configuration. The final parameter configuration identified during the tuning process is shown in Table 6.5. Since this configuration prioritizes solution quality, we denote it as *strong*. Based on observations from the tuning process, we additionally propose a *fast* configuration (Table 6.5), which aims to reduce the overall runtime. A key difference of the fast configuration is that it performs only a single mutation. During the final level of the multilevel hierarchy, when only a single individual remains, one mutation is applied to this individual. Our experiments indicate that this significantly improves solution quality compared to omitting mutation entirely.

Table 6.5: Configurations of MEMHEIPA

Parameter	*Strong*	*Fast*
Starting Pop. Size $\|POP_{l_{max}}\|$	200	10
Final Pop. Size $\|POP_0\|$	3	1
Num. Crossovers θ	10	1
Num. Parents p	2	2
Tournament Size λ	40	3
Shrinking function s	quadratic	quadratic
Mutation Percentile $M_{\%}$	0.1	final level only
Mutation Rate M_r	0.75	1.0

6.3.8 Scalability

Lastly, we investigate the scalability of our algorithm, since our implementation supports different numbers of CPU threads. The population is distributed among these threads, and each thread independently launches GPU kernels to process its assigned individuals. To evaluate the scalability of this approach, we execute the algorithm using one, two, four, eight and sixteen CPU threads and compare the resulting speedups. The results are shown in Figure 6.7. Increasing the number of CPU threads generally reduces the runtime. However, the achieved speedup is clearly sublinear. Using two or four CPU threads, for example, does not result in a corresponding twofold or fourfold reduction in runtime. Furthermore, increasing the number of CPU threads beyond four does not lead to a noticeable improvement and using sixteen threads even leads to slower running times than using four threads. One possible explanation is that the GPU cannot execute all kernels submitted by the different CPU threads concurrently, and might become overloaded in the case of sixteen threads. Furthermore, not all parts of the algorithm can be parallelized, which also limits the achievable speedup.

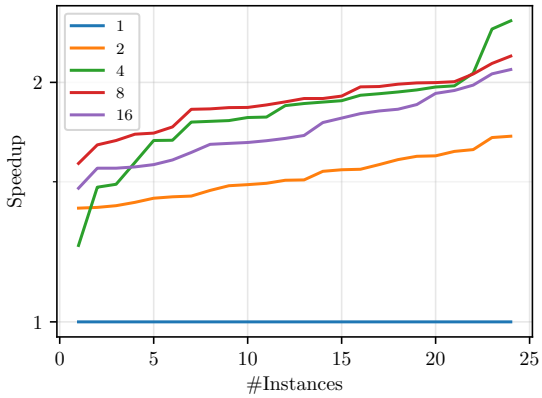


Figure 6.7: Comparison of different numbers of CPU threads.

6.4 Comparison to SOTA

Since our main goal is to improve the solution quality achieved by existing GPU algorithms, we compare MEMHEIPA against state-of-the-art graph partitioners. Our primary reference algorithm is JET in its preconfigurations *default* and *ultra*. To enable a fair comparison, we utilize the same benchmark set that JET was evaluated on (depicted in Table 6.2). Furthermore, we also choose an imbalance of $\epsilon = 3\%$. We extend the set of k-values to $\{2, 4, 8, 16, 32, 64, 128, 256\}$, in order to evaluate the algorithms across a broader range of problem instances. In contrast, the authors of JET only considered $k \in \{32, 64, 128, 256\}$. Each parameter configuration is executed five times, and we report the average runtime and solution quality over these runs. To provide an additional reference point, we compare against G-KWAY [33], another GPU multilevel graph partitioner. G-KWAY was not able to execute 12 out of 65 instances, and also was not able to partition any graph for $k \in \{128, 256\}$. The implementations of the algorithms proposed by Godarzi et al. [32] and Fagginer Auer [14] were not publicly available, preventing a direct comparison. After succesfully tuning our algorithm we have derived two configurations, denoted as *strong* and *fast* (shown in Table 6.5). For brevity, we refer to our algorithm in the fast configuration as MEMHEIPA-F and in the strong configuration as MEMHEIPA-S.

Traditionally, CPU partitioners have achieved a higher solution quality compared to GPU approaches. To investigate whether MEMHEIPA can close this gap, we additionally compare MEMHEIPA to state-of-the-art CPU algorithms. To this end, we chose METIS and KAFFPA as baselines. METIS is a widely used and well known partitioner, and KAFFPA is the current state-of-the-art for CPU partitioners. We use KAFFPA in its preconfiguration *strong*, which we refer to as KAFFPA-S for brevity.

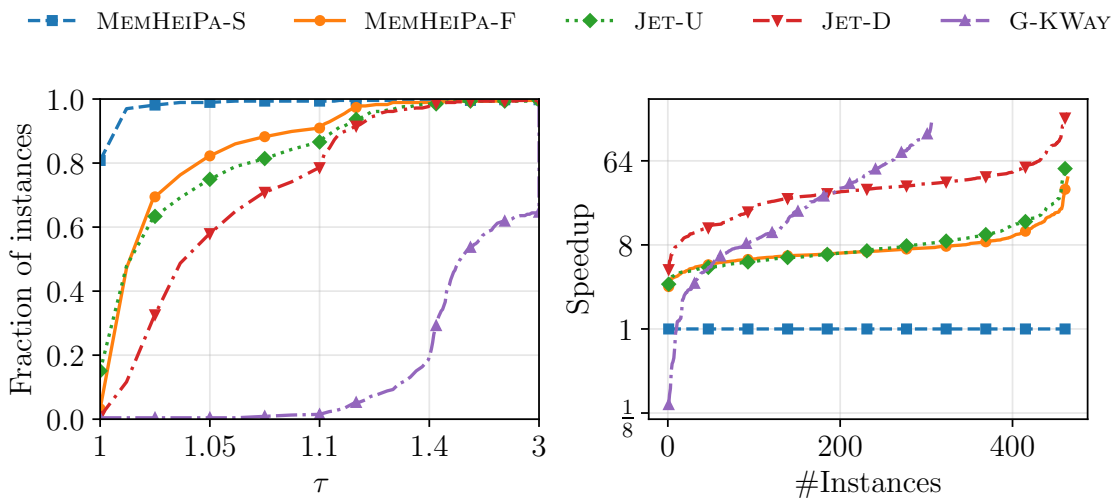


Figure 6.8: Solution quality (left) and speedup over MemHeiPa-S (right) for the various SOTA GPU algorithms.

6.4.1 GPU-Algorithms

Figure 6.8 presents a performance profile and speedup plot comparing MemHeiPa with Jet and G-kway. MemHeiPa-S clearly achieves the highest solution quality among all GPU algorithms, finding the best solution on 80 % of instances. Jet-Ultra finds the best solution on 15 % of instances, followed by MemHeiPa-F with 3 %. Both Jet-Default and G-kway achieve the best solution on less than 1 % of instances. Across the majority of performance ratios, the performance profile reveals a clear ranking: MemHeiPa-S achieves the best solution quality, followed by MemHeiPa-F, Jet-Ultra, Jet-Default, and G-kway. This ranking is further supported by the geometric mean edge cuts achieved by the algorithms. Compared to MemHeiPa-S, the average edge cuts achieved by MemHeiPa-F, Jet-Ultra, Jet-Default, and G-kway are 3.4 %, 4.5 %, 7 %, and 62 % higher, respectively. Further note that the performance of G-kway is affected by robustness issues, as it was only able to produce partitions on roughly 65 % of instances, of which 25 % violated the balance constraint.

Regarding execution time, MemHeiPa-S is the slowest algorithm, followed by MemHeiPa-F, Jet-Ultra, G-kway and Jet-Default. Relative to MemHeiPa-S, the geometric mean speedups achieved by MemHeiPa-F, Jet-Ultra, G-kway, and Jet-Default are 7.19×, 7.68×, 16.3×, and 29.65×, respectively. This difference in runtime is expected, as MemHeiPa-S performs extensive evolutionary search steps and iterated multilevel schemes. Moreover, the differences between Jet-Ultra and MemHeiPa-F are negligible, with the median absolute difference in running time amounting to 0.21 seconds. Overall, MemHeiPa-S substantially improves the solution quality of state-of-the-art GPU graph partitioners, at the cost of increased runtime, whereas MemHeiPa-F outper-

forms JET-ULTRA with respect to solution quality while offering similar running times.

Additionally, we investigate how the number of blocks affects the performance of the algorithms. For this we distinguish between small and large values of k. We define small k-values as $k \in \{2, 4, 8, 16\}$ and large k-values as $k \in \{32, 64, 128, 256\}$. Figure 6.9 shows the results. Interestingly, the advantage of MEMHEIPA becomes more pronounced for smaller values of k. Both the differences in solution quality between MEMHEIPA and the other GPU algorithms are larger, and the speedups achieved by the other algorithms over MEMHEIPA-S are smaller. For example, JET-ULTRA is on average 6.18× faster than MEMHEIPA-S for small k-values, compared to 9.54× for large k-values. In particular, we want to highlight the comparison between MEMHEIPA-F and JET-ULTRA: For small

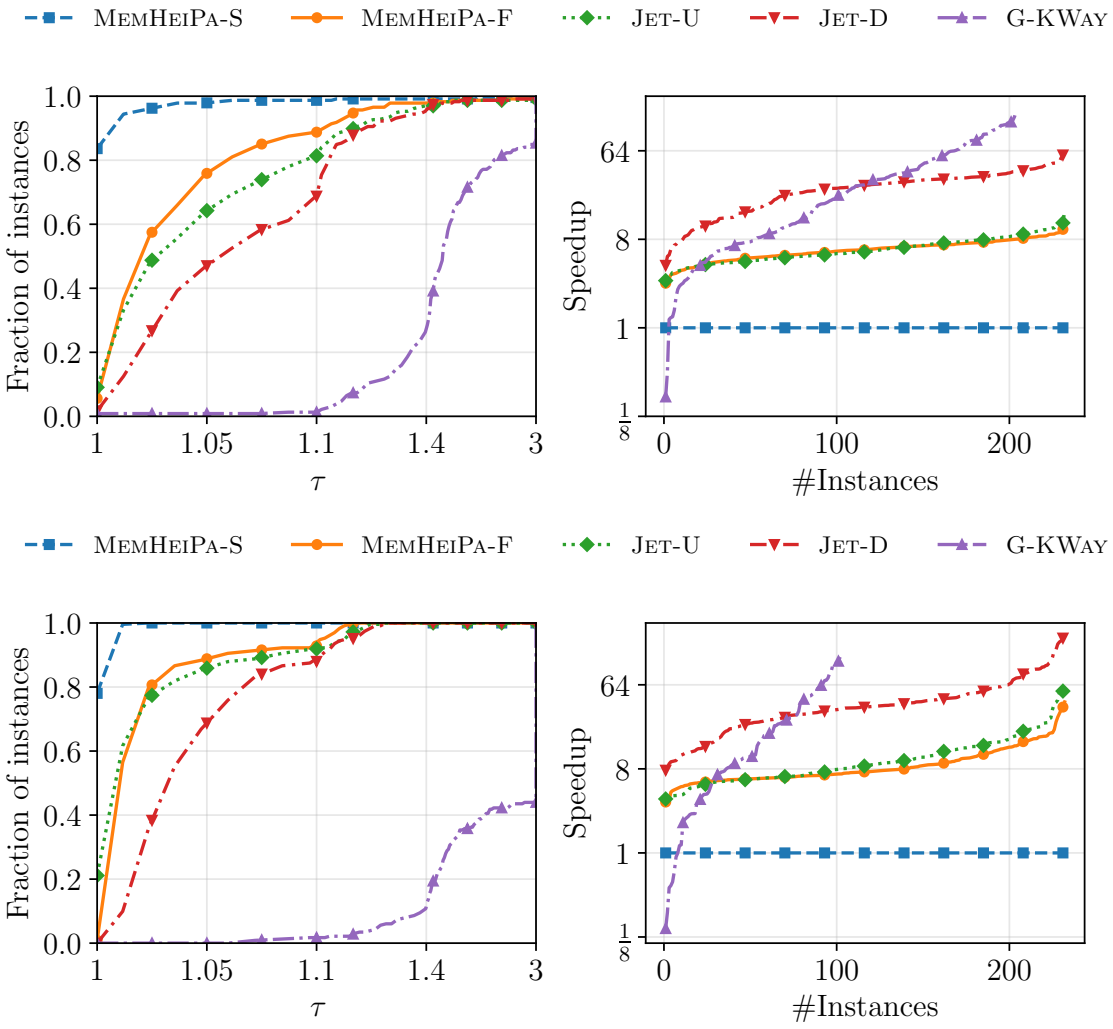


Figure 6.9: Comparison to SOTA GPU algorithms split into small k-values (figure at the top) and large k-values (figure at the bottom).

k-values, the geometric mean edge cut achieved by MEMHEIPA-F is 1.94% lower than the geometric mean edge cut achieved by JET-ULTRA. At the same time, their average running time is about equal, with their geometric mean speedups over MEMHEIPA-S being 6.14× and 6.18×, respectively. Thus, especially for small k-values, MEMHEIPA-F is able to outperform JET-ULTRA.

6.4.2 CPU-Algorithms

To investigate whether MEMHEIPA can narrow the solution-quality gap between CPU and GPU graph partitioners, we compare MEMHEIPA against the state-of-the-art partitioner KAFFPA-S and the well-established partitioner METIS. Since KAFFPA-S requires substantial time to compute high-quality solutions, we impose an 8-hour time limit and restrict the comparison to instances that KAFFPA-S solves within this limit. The results are presented in Figure 6.10.

Regarding solution quality, KAFFPA-S finds the best solution on 51.7% of instances, followed by MEMHEIPA-S with 35.8% and JET-ULTRA with 10.5%. On 35.5% of instances, MEMHEIPA-S achieves a strictly lower edge cut than all other algorithms. On this subset of instances, the geometric mean edge cut of MEMHEIPA-S is 5.79% lower than that of KAFFPA-S and 4.3% lower than that of JET-ULTRA. The largest improvement over KAFFPA-S occurs on `indochina-2004` with $k = 2$, where MEMHEIPA-S achieves a 71.1% lower edge cut. On the remaining 64.5% of instances, where MEMHEIPA-S does not achieve the best solution, its geometric mean edge cut is 8.9% higher than that of KAFFPA-S, but still 3.78% lower than that of JET-ULTRA. Across the entire set of instances, the geometric mean edge cuts of MEMHEIPA-S and JET-ULTRA are 3.5%

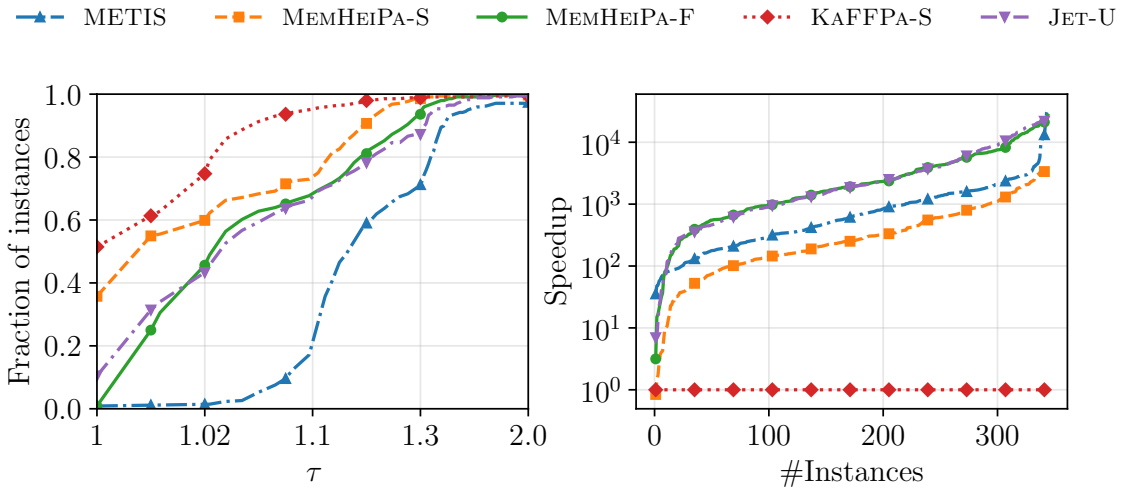


Figure 6.10: Solution quality (left) and speedup over KAFFPA (right) for the various CPU and GPU algorithms.

and 7.7% higher than that of KAFFPA-S, respectively. However, MEMHEIPA-S retains a substantial runtime advantage: it is on average 254× faster than KAFFPA-S. Thus, MEMHEIPA-S substantially narrows the solution-quality gap to the CPU state-of-the-art while retaining the large speed advantage of GPU-based partitioners. Furthermore, it can be seen that MEMHEIPA-F consistently finds lower edge cuts than METIS, with a 13.4% lower geometric mean edge cut. Additionaly, MEMHEIPA-F is faster than METIS, achieving a geometric mean speedup of $3.6\times$. Consequently, MEMHEIPA-F outperforms METIS in both solution quality and execution time.

6.5 CPU+GPU Partitioner

Finally, we evaluate our hybrid partitioner presented in Chapter 5, which integrates MEMHEIPA into KAFFPAE. To this end, we make use of the Walshaw benchmark, which KAFFPAE was evaluated on. The Walshaw benchmark consists of the graph instances depicted in Table 6.3 and $k \in \{2, 4, 8, 16, 32, 64\}$. Regarding the imbalance parameter, we limit our experiments to $\epsilon = 3\%$. The program is run with 16 MPI ranks and the time limit is set to 25 minutes. We start our experimental evaluation with a comparison of the two different strategies for initializing the population of the GPU-thread. The first approach we highlighted, the *Copy-Model*, initializes the threads population as a copy of the PEs population. The second approach, the *Producer-Consumer-Model*, makes the GPU-thread start with an empty population. Additionally, we investigate the effect of using different amounts of GPUs. We compare using 1, 2 and 4 GPUs, since 4 is the hardware limit of our machine. After determining the configuration which achieves the highest quality, we compare our hybrid algorithm to KAFFPAE in its standard form. We conclude this section with an additional comparison to KAFFPA-S and METIS. For brevity we refer to our hybrid algorithm as KAFFPAE+GPU.

6.5.1 Population Initialization Models

The two population initialization models represent different tradeoffs between exploiting existing solutions and introducing additional diversity. Since the Copy-Model initializes the GPU-thread with a copy of the PE's population, the thread can focus on refining existing individuals using V-cycles. In contrast, the Producer-Consumer-Model initializes the GPU-thread with an empty population, allowing it to introduce additional diversity by generating new individuals independently. Figure 6.11 compares both models using a convergence plot (left) and performance profile (right) to determine which approach achieves better results. The differences between the models are small, with both achieving similar convergence rates and fractions of best instances. Nevertheless, the Copy-Model shows a slight advantage: its convergence rate is marginally faster, and it outperforms the Producer-Consumer-Model across all performance ratios. Since KAFFPAE is executed with 16 PEs, each maintaining and refining its own local subpopulation, the search process may already

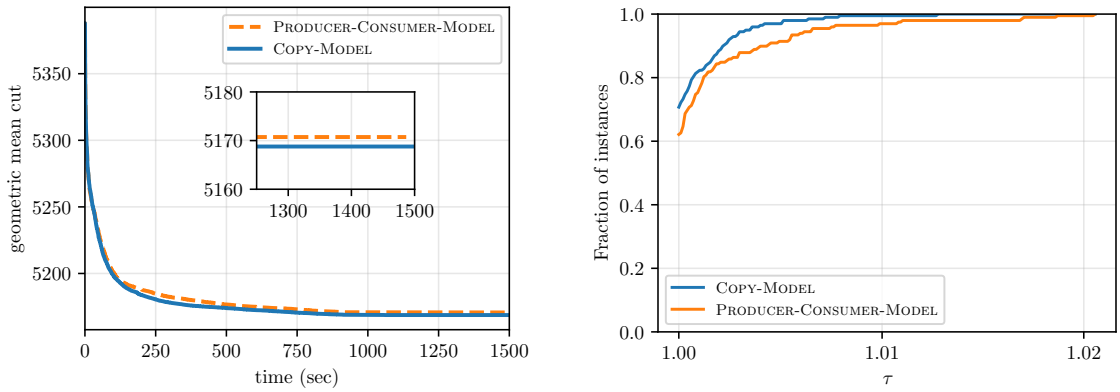


Figure 6.11: Convergence plot (left) and performance profile (right) comparing the two different population initialization models. The performance profile compares the edge cut achieved at the end of the runtime.

benefit from sufficient diversity. Therefore, further improving existing individuals appears to be more beneficial than introducing additional diversity. Hence, we use the copy model for the following experiments.

6.5.2 Amount of GPU devices

Next, we investigate how the number of available GPU devices affects the performance of KaFFPaE+GPU. Since each GPU is dedicated to a single PE, additional GPUs allow more PEs to refine their local populations using both a GPU-thread and a CPU-thread. Figure 6.12 shows that increasing the number of GPUs consistently improves both the convergence rate and the final solution quality. Although the differnces are small, this still

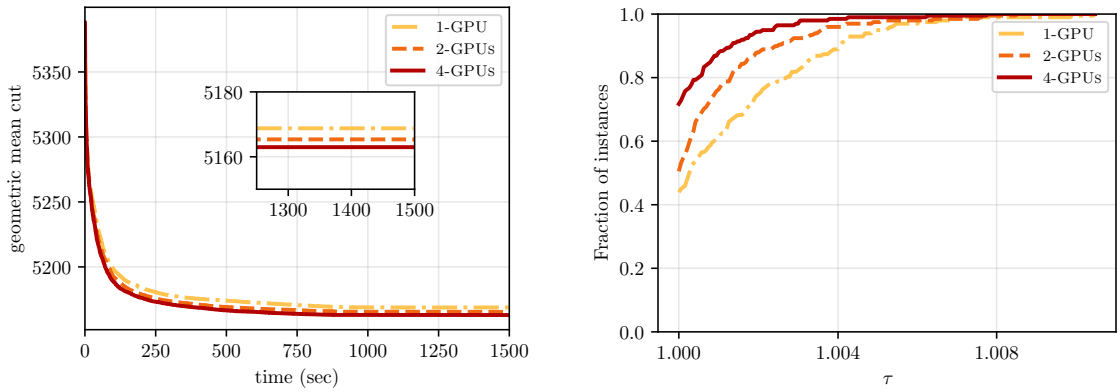


Figure 6.12: Convergence plot (left) and performance profile (right) comparing usage of different amounts of GPU devices.

indicates that the additional GPU resources are effectively utilized by the hybrid search process. While this trend is expected since additional computational resources become available, it also demonstrates that KaFFPaE+GPU scales well with the number of GPUs and is able to translate the increased computational power into higher-quality solutions.

6.5.3 Comparison to KaFFPaE without GPU

Now, we evaluate the benefit of extending KaFFPaE with GPU acceleration by comparing the original CPU version of KaFFPaE with our hybrid algorithm using four GPUs and the *Copy-Model*. Figure 6.13 reveals, that KaFFPaE+GPU converges substantially faster than the CPU-only version: The geometric mean edge cut achieved by KaFFPaE+GPU after 100 seconds is already lower than the geometric mean edge cut achieved by KaFFPaE after 25 minutes. Furthermore, the final solutions found by both algorithms differ significantly. KaFFPaE+4GPUs finds the best solution on 80% of instances, whereas KaFFPaE achieves the best solution on only 42% of instances. In addition, the edge cuts produced by KaFFPaE are up to 5% higher than those found by KaFFPaE+4GPUs. These results highlight the benefit of our CPU-GPU partitioning approach, which is able to achieve substantially higher solution quality within the same runtime budget. More broadly, they demonstrate the potential of algorithms that exploit all available hardware resources of modern heterogeneous systems.

6.5.4 Comparison with CPU Partitioners

Lastly, we compare our hybrid partitioner with the established CPU partitioners KaFFPa-S and Metis to assess its performance relative to standard CPU-based approaches. The results are presented in Figure 6.14. Regarding runtime, both Metis and KaFFPa-S achieve considerable speedups over KaFFPaE. This is expected, as both algorithms are

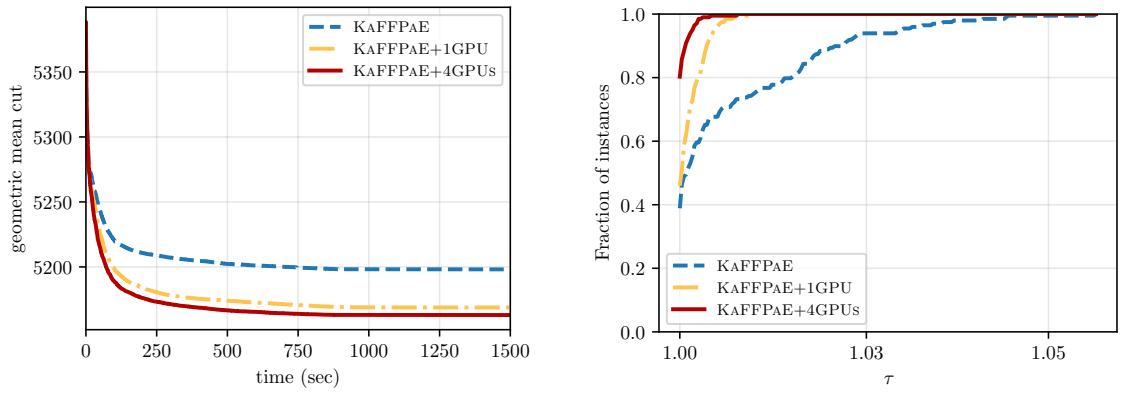


Figure 6.13: Convergence plot (left) and performance profile (right) comparing KaFFPaE with KaFFPaE+GPU.

executed only once, whereas KaFFPaE and the hybrid approaches are given 25 minutes to search for high-quality solutions. Metis is particularly fast, requiring only 0.02 seconds on average to compute a solution, while KaFFPa-S requires 2.28 seconds. Regarding solution quality, the hybrid approaches clearly outperform KaFFPa-S and Metis. KaFFPaE+4GPUs finds the best solution on 80.3% of instances, followed by KaFFPaE+GPU with 46% and KaFFPaE with 38%. In comparison, KaFFPa-S and Metis find the best solution on only 6.1% and 1.5% of instances, respectively. This difference is also reflected in the geometric mean edge cut: compared to KaFFPaE+4GPUs, KaFFPa-S and Metis achieve edge cuts that are 4.69% and 18.34% higher, respectively. Overall, these results demonstrate that the proposed hybrid approach is not only able to noticeably improve upon the original CPU-based KaFFPaE, but also achieves substantially higher-quality partitions than established CPU partitioners.

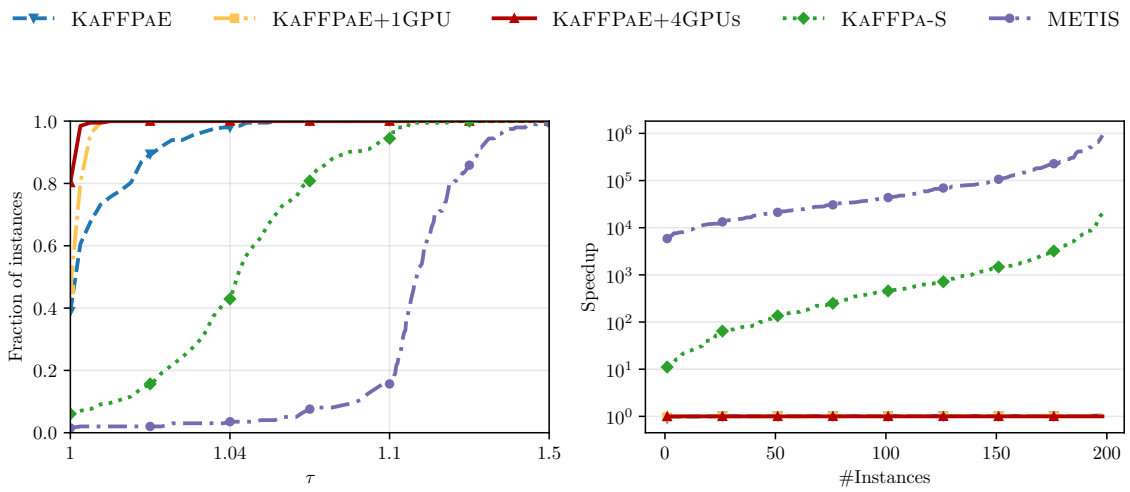


Figure 6.14: Comparison to different CPU algorithms

CHAPTER 7

Discussion

7.1 Conclusion

This work addressed the graph partitioning problem, which seeks to divide a graph into k roughly equal-sized blocks while minimizing the number of cut edges. We proposed a novel multilevel memetic GPU graph partitioner that substantially improves upon existing GPU approaches in terms of solution quality. Our algorithm maintains a population of candidate partitions throughout the multilevel hierarchy, which is iteratively improved by evolutionary operators. A crossover operator preserves promising vertex groupings shared among parent solutions, while a mutation operator based on iterated multilevel schemes further improves high-quality individuals. Our *strong* configuration achieves edge cuts that are on average 4.5 % lower than those of JET-ULTRA, while the *fast* configuration outperforms METIS in both solution quality and runtime. These results highlight the potential of memetic search to enhance the solution quality of GPU graph partitioners.

Furthermore, an increasing number of HPC systems are equipped with both CPUs and GPUs, while most graph partitioners still target either architecture exclusively. This leaves part of the available computational resources unused and motivates the development of hybrid partitioners. We therefore derived a hybrid CPU+GPU partitioner, by extending the distributed evolutionary algorithm KAFFPAE to additionally utilize GPUs. During the phase in which a PE improves its local population, an additional worker thread is spawned to concurrently refine the population using MEMHEIPA on the GPU. The resulting algorithm substantially accelerates convergence compared to the original CPU implementation and produces higher-quality partitions in the same timeframe. In particular, the geometric mean edge cut achieved by KAFFPAE+GPU after 100 seconds is already lower than the geometric mean edge cut achieved by KAFFPAE after 25 minutes. These results highlight the potential of heterogeneous CPU+GPU approaches for graph partitioning.

7.2 Future Work

While our algorithm outperforms JET in solution quality, it is still slower, leaving room for approaches that can improve both quality and runtime. Since the refinement algorithm accounts for the largest share of the runtime, future work could investigate ways to accelerate this component without compromising solution quality. It would also be interesting to investigate whether alternative crossover and mutation operators can further improve the quality–runtime tradeoff. Moreover, since this work demonstrates the potential of memetic GPU algorithms, an interesting direction for future research is to investigate whether similar approaches can be successfully applied to other optimization problems.

Zusammenfassung

Das Graphpartitionierungsproblem zielt darauf ab, einen Graphen in k möglichst gleich große Blöcke zu unterteilen und dabei die Anzahl der Kanten zwischen verschiedenen Blöcken zu minimieren. Aufgrund seiner zahlreichen Anwendungen besteht ein hoher Bedarf an schnellen und qualitativ hochwertigen Partitionierungsalgorithmen. Obwohl GPU-Computing in jüngster Zeit signifikante Erfolge verzeichnen konnte, erreichen GPU-basierte Graphpartitionierer noch nicht die gleiche Lösungsqualität wie CPU-basierte Algorithmen. Wir begegnen diesem Problem mit MEMHEIPA, dem ersten GPU-basierten memetischen Mehrschicht Graphpartitionierer. Unser Algorithmus verwaltet eine Population von Partitionen, die iterativ durch evolutionäre Operatoren verbessert wird. Unser Kombinationsoperator identifiziert vielversprechende Knotenmengen, die jeweils in allen Eltern zusammengruppiert sind, während unser Mutationsoperator qualitativ hochwertige Individuen weiter verbessert, mithilfe von iterierten Mehrschichtverfahren. Darüber hinaus stellen wir den ersten hybriden CPU+GPU-Partitionierer vor, der die Ressourcen heterogener Systeme vollständig ausnutzen kann. Hierzu erweitern wir den verteilten evolutionären Algorithmus KAFFPAE um MEMHEIPA. Umfangreiche Experimente zeigen, dass MEMHEIPA die Lösungsqualität bestehender GPU-basierter Graphpartitionierer deutlich verbessert. Verglichen mit dem Stand der Technik unter den GPU-basierten Partitionieren, erzielt MEMHEIPA in der Konfiguration *strong* Kantenschnitte die im Durchschnitt um 4.5% leichter sind. Des Weiteren, übertrifft unsere Konfiguration *fast* den weit verbreiteten CPU-Partitionierer METIS sowohl hinsichtlich der Laufzeit als auch der Lösungsqualität. Unser hybrider Algorithmus KAFFPAE+GPU konvergiert nicht nur deutlich schneller als die ursprüngliche CPU-implementierung, sondern erzielt mit derselben Laufzeit Partitionen mit einem geringeren Kantenschnitt.

Bibliography

[1] Kirk Schloegel, George Karypis, and Vipin Kumar. *Graph Partitioning for High Performance Scientific Simulations*. Tech. rep. TR 00-018. Minneapolis, MN, USA: Department of Computer Science and Engineering, University of Minnesota, Mar. 2000.

[2] Andrew B. Kahng et al. *VLSI Physical Design - From Graph Partitioning to Timing Closure*. Springer, 2011.

[3] Rolf H. Möhring et al. "Partitioning graphs to speedup Dijkstra's algorithm". In: *ACM J. Exp. Algorithmics* 11 (Feb. 2007), 2.8–es.

[4] Aydin Buluç et al. "Recent Advances in Graph Partitioning". In: *Algorithm Engineering - Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Lecture Notes in Computer Science. 2016, pp. 117–158.

[5] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. "Some Simplified NP-Complete Problems". In: *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*. Ed. by Robert L. Constable et al. ACM, 1974, pp. 47–63.

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

[7] Thang Nguyen Bui and Curt Jones. "Finding Good Approximate Vertex and Edge Partitions is NP-Hard". In: *Inf. Process. Lett.* 42.3 (1992), pp. 153–159.

[8] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (1998), pp. 359–392.

[9] George Karypis and Vipin Kumar. "Multilevel k-way Partitioning Scheme for Irregular Graphs". In: *J. Parallel Distributed Comput.* 48.1 (1998), pp. 96–129.

[10] Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. LNCS. Springer, 2013, pp. 164–175.

[11] Xuanhua Shi et al. "Graph Processing on GPUs: A Survey". In: *ACM Comput. Surv.* 50.6 (2018), 81:1–81:35.

[12] John Nickolls and William J. Dally. “The GPU Computing Era”. In: *IEEE Micro* 30.2 (2010), pp. 56–69.

[13] Bahareh Goodarzi, Martin Burtscher, and Dhrubajyoti Goswami. “Parallel Graph Partitioning on a CPU-GPU Architecture”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 58–66.

[14] Bastiaan Olijve Fagginger Auer. “GPU Acceleration of Graph Matching, Clustering, and Partitioning”. Doctor of Philosophy. Utrecht University, 2013.

[15] Michael S. Gilbert et al. “Jet: Multilevel Graph Partitioning on Graphics Processing Units”. In: *SIAM J. Sci. Comput.* 46.5 (2024), p. 700.

[16] TOP500. *TOP500 Highlights – June 2025*. `https://www.top500.org/lists/top500/2025/06/highs/`. Accessed: 2026-06-01. 2025.

[17] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing, Second Edition*. Natural Computing Series. Springer, 2015.

[18] David B. Fogel, ed. *Evolutionary Computation: The Fossil Record*. Piscataway, NJ: IEEE Press, 1998.

[19] Pablo Moscato. *On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms*. Tech. rep. California Institute of Technology, Caltech Concurrent Computation Program (C3P), 1989.

[20] Christian Trott et al. “The Kokkos Ecosystem: Comprehensive Performance Portability for High Performance Computing”. In: *Computing in Science Engineering* 23.5 (2021), pp. 10–18.

[21] Ümit V. Çatalyürek et al. “More Recent Advances in (Hyper)Graph Partitioning”. In: *ACM Comput. Surv.* 55.12 (2023), 253:1–253:38.

[22] Richard V. Southwell. “Stress-Calculation in Frameworks by the Method of “Systematic Relaxation of Constraints””. In: *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* 151.872 (1935), pp. 56–95.

[23] Stephen T. Barnard and Horst D. Simon. “A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems”. In: *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, PP 1993, Norfolk, Virginia, USA, March 22-24, 1993*. Ed. by Richard F. Sincovec et al. SIAM, 1993, pp. 711–718.

[24] Ralf Diekmann et al. “Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM”. In: *Parallel Comput.* 26.12 (2000), pp. 1555–1581.

[25] Brian W. Kernighan and Shen Lin. “An efficient heuristic procedure for partitioning graphs”. In: *Bell Syst. Tech. J.* 49.2 (1970), pp. 291–307.

[26] Charles M. Fiduccia and Robert M. Mattheyses. “A linear-time heuristic for improving network partitions”. In: *Proceedings of the 19th Design Automation Conference, DAC ’82, Las Vegas, Nevada, USA, June 14-16, 1982*. Ed. by James S. Crabbe, Charles E. Radke, and Hillel Ofek. ACM/IEEE, 1982, pp. 175–181.

[27] Fred W. Glover and Manuel Laguna. *Tabu Search*. Kluwer, 1997.

[28] Una Benlic and Jin-Kao Hao. “An effective multilevel tabu search approach for balanced graph partitioning”. In: *Comput. Oper. Res.* 38.7 (2011), pp. 1066–1075.

[29] Philippe Galinier, Zied Boujbel, and Michael Coutinho Fernandes. “An efficient memetic algorithm for the graph partitioning problem”. In: *Ann. Oper. Res.* 191.1 (2011), pp. 1–22.

[30] George Karypis and Vipin Kumar. “Multilevel k-way Hypergraph Partitioning”. In: *VLSI Design* 11.3 (2000), p. 019436.

[31] Usha N. Raghavan, Réka Albert, and Soundar Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: *Physical Review E* 76.3 (2007), p. 036106.

[32] Bahareh Goodarzi et al. “High Performance Multilevel Graph Partitioning on GPU”. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 2019, pp. 769–778.

[33] Wan-Luan Lee et al. “G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner”. In: *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024*. Ed. by Vivek De. ACM, 2024, 105:1–105:6.

[34] Dominique Lasalle and George Karypis. “Multi-threaded Graph Partitioning”. In: *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. IEEE Computer Society, 2013, pp. 225–236.

[35] Michael S. Gilbert et al. “Performance-Portable Graph Coarsening for Efficient Multilevel Graph Analysis”. In: *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 2021, pp. 213–222.

[36] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *J. Parallel Distributed Comput.* 74.12 (2014), pp. 3202–3216.

[37] Jin Kim et al. “Genetic approaches for graph partitioning: a survey”. In: *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*. Ed. by Natalio Krasnogor and Pier Luca Lanzi. ACM, 2011, pp. 473–480.

[38] Alan J. Soper, Chris Walshaw, and Mark Cross. “A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning”. In: *J. Glob. Optim.* 29.2 (2004), pp. 225–241.

[39] Chris Walshaw and Mark Cross. “JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview”. In: *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Ed. by F. Magoulès. Civil-Comp Ltd., 2007, pp. 27–58.

[40] Una Benlic and Jin-Kao Hao. “A Multilevel Memetic Approach for Improving Graph k-Partitions”. In: *IEEE Trans. Evol. Comput.* 15.5 (2011), pp. 624–642.

[41] Dan Gusfield. “Partition-distance: A problem and class of perfect graphs arising in clustering”. In: *Inf. Process. Lett.* 82.3 (2002), pp. 159–164.

[42] Peter Sanders and Christian Schulz. “Distributed Evolutionary Graph Partitioning”. In: *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*. Ed. by David A. Bader and Petra Mutzel. SIAM / Omnipress, 2012, pp. 16–29.

[43] Jens Maue and Peter Sanders. “Engineering Algorithms for Approximate Weighted Matching”. In: *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*. Ed. by Camil Demetrescu. Lecture Notes in Computer Science. Springer, 2007, pp. 242–255.

[44] JR L. R. FORD and Delbert Ray Fulkerson. “Maximal Flow Through a Network”. In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.

[45] Daniel Delling et al. “Graph Partitioning with Natural Cuts”. In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. IEEE, 2011, pp. 1135–1146.

[46] Nicholas Wilt. *The CUDA handbook. a comprehensive guide to GPU programming*. eng. Upper Saddle River, NJ: Addison-Wesley, 2013, 1 online resource (1 v.)

[47] Jason Sanders. *CUDA by example. an introduction to general-purpose GPU programming*. eng. Ed. by Edward [MitwirkendeR] Kandrot. Upper Saddle River, N.J.: Addison-Wesley, 2010, 1 online resource (xix, 290 p.)

[48] Michel Toulouse, Krishnaiyan Thulasiraman, and Fred W. Glover. “Multi-level Cooperative Search: A New Paradigm for Combinatorial Optimization and an Application to Graph Partitioning”. In: *Euro-Par ’99 Parallel Processing, 5th International Euro-Par Conference, Toulouse, France, August 31 - September 3, 1999, Proceedings*. Ed. by Patrick Amestoy et al. Lecture Notes in Computer Science. Springer, 1999, pp. 533–542.

[49] Chris Walshaw. “Multilevel Refinement for Combinatorial Optimisation Problems”. In: *Ann. Oper. Res.* 131.1-4 (2004), pp. 325–372.

[50] Elizabeth D. Dolan and Jorge J. Moré. “Benchmarking Optimization Software with Performance Profiles”. In: *Mathematical Programming* 91 (2002), pp. 201–213.

[51] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83.

[52] David A. Bader et al. “Benchmarking for Graph Clustering and Partitioning”. In: *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Ed. by Reda Alhajj and Jon G. Rokne. Springer, 2018.

[53] Michael Gilbert et al. *Graph Dataset for "Jet: Multilevel Graph Partitioning on Graphics Processing Units"*. 2024.

[54] Timothy A. Davis and Yifan Hu. “The university of Florida sparse matrix collection”. In: *ACM Trans. Math. Softw.* 38.1 (2011), 1:1–1:25.

[55] Weihua Hu et al. “Open Graph Benchmark: Datasets for Machine Learning on Graphs”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020.

[56] Paolo Boldi and Sebastiano Vigna. “The webgraph framework I: compression techniques”. In: *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*. Ed. by Stuart I. Feldman et al. ACM, 2004, pp. 595–602.

[57] Paolo Boldi et al. “Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks”. In: *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. Ed. by Sadagopan Srinivasan et al. ACM, 2011, pp. 587–596.

[58] Jure Leskovec et al. “Kronecker Graphs: An Approach to Modeling Networks”. In: *J. Mach. Learn. Res.* 11 (2010), pp. 985–1042.

[59] Stephen J. Wright. “Coordinate descent algorithms”. In: *Math. Program.* 151.1 (2015), pp. 3–34.